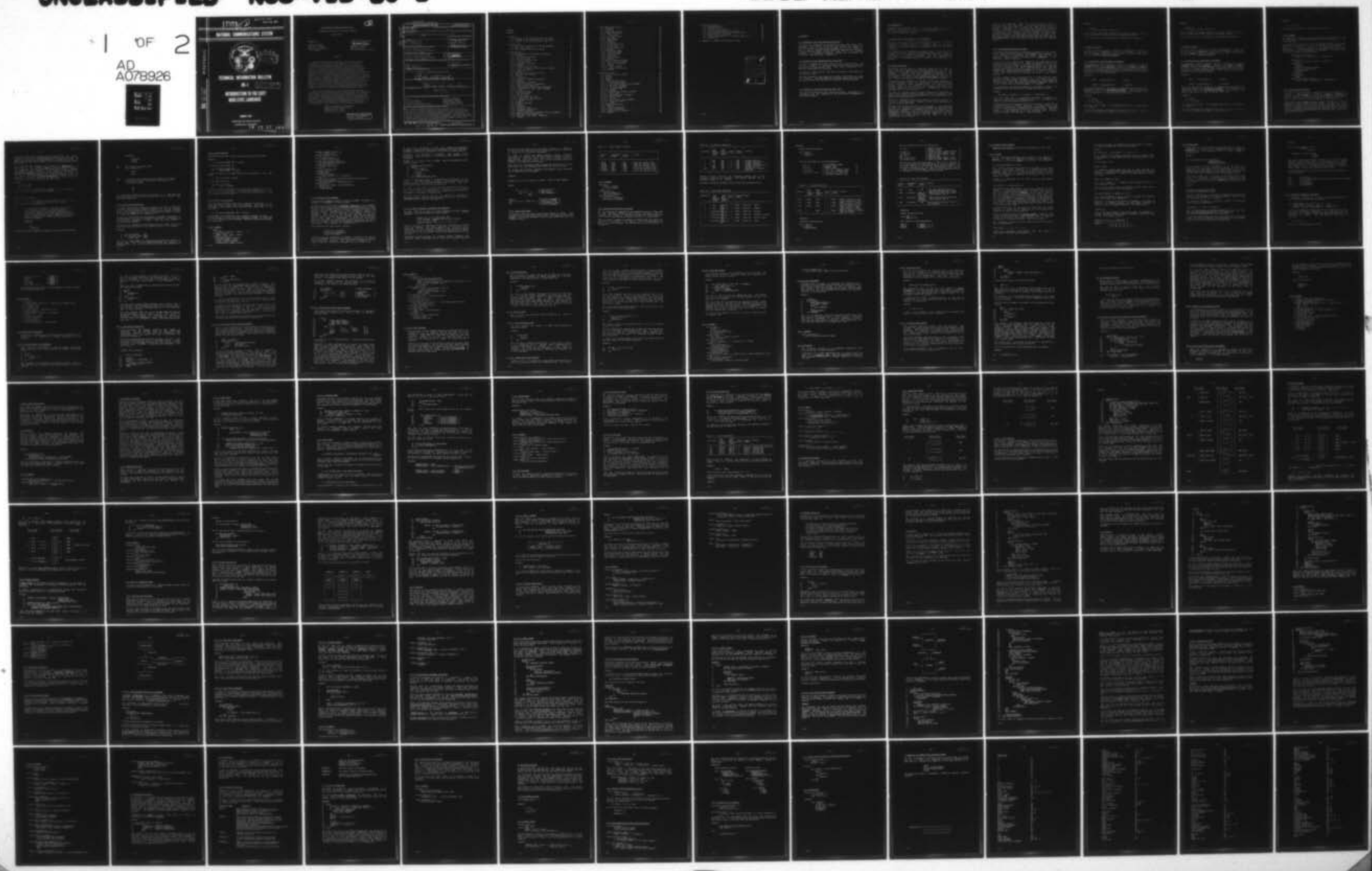
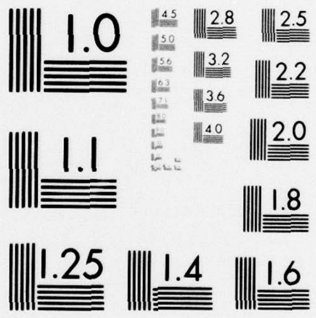
AD-A078 926 NATIONAL COMMUNICATIONS SYSTEM WASHINGTON DC F/G 9/2
INTRODUCTION TO THE CCITT HIGH LEVEL LANGUAGE.(U)
JAN 80 R M FENICHEL
UNCLASSIFIED NCS-TIB-80-1 SBIE-AD-E100 320 NL

1 OF 2
AD
A078926




1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

NCS TIB 80-1

NATIONAL COMMUNICATIONS SYSTEM

TECHNICAL INFORMATION BULLETIN

80-1

INTRODUCTION TO THE CCITT HIGH LEVEL LANGUAGE

JANUARY 1980

APPROVED FOR PUBLIC RELEASE;

DISTRIBUTION UNLIMITED

2

NCS TECHNICAL INFORMATION BULLETIN 80-1

INTRODUCTION TO THE CCITT HIGH LEVEL LANGUAGE

January 1980

PREPARED BY:

ROBERT M. FENICHEL
Electronics Engineer
NCS Office of Technology
and Standards

APPROVED FOR PUBLICATION:

Marshall L. Cain

MARSHALL L. CAIN
Assistant Manager
NCS Technology and Standards

FOREWORD

Among the responsibilities assigned to the Manager, National Communications System (NCS), is the management of the Federal Telecommunication Standards Program, which is an element of the overall Federal Standardization Program. Under this program, the NCS, with the assistance of the Federal Telecommunication Standards Committee, identifies, develops and coordinates proposed Federal Standards which either contribute to the interoperability of functionally similar Federal telecommunications equipment and networks or to the achievement of an efficient interface between the information processing function and telecommunication networks. In developing and coordinating these standards, considerable effort is expended in initiating and pursuing joint standards development efforts with appropriate technical committees of the Electronic Industries Association, the American National Standards Institute, the International Organization for Standardization, and the International Telegraph and Telephone Consultative Committee (CCITT) of the International Telecommunications Union.

This Technical Information Bulletin (TIB) reprints the manual "Introduction to CHILL" written by CCITT Working Party XI/3 (Contribution Comm XI-342E, October 1979), and provides an introduction to the high level language standard being developed by CCITT for telecommunication applications. The TIB has been published to inform interested Federal activities of the progress of this international standards effort. Any comments, inputs, or statements of requirement which could assist in the advancement of this work are welcome and should be addressed to:

Office of Technology and Standards
National Communications System
Washington, D.C. 20305
Telephone (202) 692-2124

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER 14 NCS-TIB-80-1	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 Introduction to the CCITT High Level Language.		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Robert M. Fenichel		8. CONTRACT OR GRANT NUMBER(s) 12 97
9. PERFORMING ORGANIZATION NAME AND ADDRESS National Communications System Office of Technology and Standards Washington, D.C. 20305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS National Communications System Office of Technology and Standards Washington, D.C. 20305		12. REPORT DATE 11 Jan[redacted]1980
		13. NUMBER OF PAGES 96
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) 18 SBIE		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution unlimited; approved for public release.

19 AD-E100 320

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

9 Technical information bulletin,

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

CHILL	High Level Language
Circuit Switching	Programming Languages
CCITT Study Group XI	Stored Program Control
Communications Switching Center	Telephone Central Office

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This Technical Information Bulletin (TIB) reprints the third edition of "Introduction to CHILL", published by Working Party 3 of the International Telegraph and Telephone Consultative Committee's (CCITT) Study Group XI. CHILL is the name given the CCITT High Level Language. The CHILL high level language standard is being developed for programming stored program controlled telephone exchanges, but it is considered general enough for other telecommunications programming applications as well.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

410 914 [illegible]

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

CONTENTS

0. Preface

0.1. Preface to the first edition, Nov. 1977

To promote the spreading of knowledge about the CCITT High Level Language, the CCITT WP XI/3-2 has decided to produce an introductory manual. The present proposal is a first draft for such a manual. Readers are encouraged to express their comments. In particular the manual should be tried out on the intended audience. In submitting comments, please use the comment and evaluation sheet provided in Appendix A.

0.2. Preface to the second edition, June 1978

A number of comments have been received on the first edition. We are very grateful to those who have taken the effort to provide constructive criticisms and ideas. They have contributed to a definite improvement of the manual.

In addition this edition has been extended to cover most main constructs of CHILL.

Because of delays in the definition of CHILL, this manual is still not complete. The most important features lacking are concurrent processing, power sets and parts of character string handling.

0.3. Preface to the third edition, Aug. 1979

This edition includes all main features of CHILL. The manual is in accordance with the proposal for the CCITT CHILL recommendation, the Brown Document.

1. Introduction

This book is an informal introduction to the CCITT High Level Language. The language is defined in the CCITT "Brown Document" of Sept. 1979, entitled: "Proposal for a recommendation for a CCITT High Level Programming Language".

1.1. High level languages for SPC programming

With the increasing use of Stored Program Control (SPC) telephone exhanges, programming is becoming an important part of telephone technology. Software systems for SPC have become very large and complex. Thus the use of proper programming tools is of great importance.

The CCITT High Level Programming Language, CHILL, is one such tool. It has been designed to be suitable for programming all kinds of programs for SPC telephone exchanges. This application covers a wide range of computer programming. Thus the language is considered to be general enough for a number of other applications as well.

1.2. About this book

The purpose of this book is to teach the reader how to use the CCITT High Level Language in an elementary fashion. The reader is assumed to have some practical programming experience. The knowledge of another high level language is an advantage. The presentation concentrates on the language as such, not the use of its compiler or other related program development systems. The book is not an introduction to computer programmming in general.

In the presentation emphasis is placed on writing readable and reliable programs. Also the presentation tries to show typical uses of constructs, not all possible uses. In order to learn advanced usage of the language, the reader is referred to the language definition and to material accompanying particular implementations of the language.

The book is written so as to make it possible for the reader to start writing simple workable programs as soon as possible. The simple, basic features are explained first. More features and more details are explained later.

The book is intended for sequential reading from front to back. It is not a reference manual. However, to aid cross referencing an index is found at the back.

Because of the recursive nature of the language, it is impossible to make the presentation completely sequential without too much duplication. Therefore, occasional use will be found of yet unexplained material. This is only done at places where detailed knowledge of the features is not considered essential. If more information is still desired, use the index to find more explanations of the feature.

CHILL is a rich language. There may even be several ways of obtaining the same objectives. In an introduction like this, a selection of features must be made and some details must be skipped. Obviously such a selection will reflect the tastes and opinions of the author. It is envisaged that parts of this book, in particular the examples, may be replaced, reflecting other tastes and opinions. In particular, CHILL allows, in some cases, more than one way of writing constructs with the same meaning. For pedagogical reasons only one style has been used in this manual. For completeness, the equivalent styles are shown in a separate chapter.

1.3. The language description method

A language description may be considered to consist of two parts. First there is the description of the <u>syntax</u> of the language, i.e. which sequences of characters form legal constructs in the language. Second there is the description of <u>semantics</u>, i.e. the meanings of these constructs.

In this book both syntax and semantics are described fairly informally using prose and a number of examples. Most of the examples have a line number in the left hand margin. This is only for easy reference from the accompanying explanation. The line numbers are not part of the CHILL program text. For reference purposes each section contains a summary of the syntax written in a somewhat more condensed and formal form, as a set of syntactic rules. The rules used are similar to those of the "Brown Document".

Syntactic categories are indicated by one or more English words enclosed between angular brackets, e.g. <row mode>. The same denotation is used without the angular brackets in the accompanying English prose text. Characters and character sequences appearing in CHILL source text are as they appear, e.g. MODULES.

To indicate that one syntactic element is followed by another one (concatenation) they are written in order from left to right.

Example:

```
    <assignment statement>::= <location> := <expression> ;
```

This rule says that an assignment statement is composed of a location followed by the assignment symbol, ":=", followed by an expression and terminated by semicolon, ";".

Syntactic elements are grouped using curly brackets. Repetition of a group zero or more times is indicated by following the right bracket by an asterisk. Repetition one or more times is indicated by a "+". Alternatives are indicated by a vertical bar. Thus {A}* stands for any sequence of of As including none. Optional syntactic elements are enclosed in square brackets.

Example:

```
    <name>::=
        <letter>{<letter>|<digit>|_}*
```

This rule says that a name consists of a letter possibly followed by any sequence of letters, digits or underscore symbols.

When used in concert, concatenation is done before alternation.

2. Language Basics

A CHILL program logically consists of three parts. There is a description of the actions which are to be performed, a description of the objects which are manipulated by these actions and a description of the program structure.

2.1. Expressions and Assignment Statements

An expression is used to compute a value. It consists of a list of operands connected by operators. Operators are dyadic or monadic. A dyadic operator takes two operands, one to the left and one to the right. A monadic operator takes one operand to the right only. Expressions are evaluated from left to right taking into account operator priority and parentheses.

Examples:

```
        A+B*3    is equivalent to        A+(B*3)

        -A*B+C   is equivalent to        ((-A)*B)+C
```

The value delivered or yielded by an expression may be stored in a location by means of an assignment statement. The value on the right hand side of the assignment symbol ":=" is stored in the location denoted by the name on the left hand side.

Examples:

```
    1.  I:= 1;
    2.  I:= -2;
    3.  I:= (I+2)*J;
```

The first line is read: "I is assigned the value 1", or for short: "I becomes 1".

Multiple assignments are indicated by giving a list of location denotations separated by commas on the left hand side.

Example:

```
    <name>::=
        <letter>{<letter>|<digit>|_}*
```

This rule says that a name consists of a letter possibly followed by any sequence of letters, digits or underscore symbols.

When used in concert, concatenation is done before alternation.

2. Language Basics

A CHILL program logically consists of three parts. There is a description of the actions which are to be performed, a description of the objects which are manipulated by these actions and a description of the program structure.

2.1. Expressions and Assignment Statements

An expression is used to compute a value. It consists of a list of operands connected by operators. Operators are dyadic or monadic. A dyadic operator takes two operands, one to the left and one to the right. A monadic operator takes one operand to the right only. Expressions are evaluated from left to right taking into account operator priority and parentheses.

Examples:

```
        A+B*3   is equivalent to        A+(B*3)

        -A*B+C  is equivalent to        ((-A)*B)+C
```

The value delivered or yielded by an expression may be stored in a location by means of an assignment statement. The value on the right hand side of the assignment symbol ":=" is stored in the location denoted by the name on the left hand side.

Examples:

```
    1.  I:= 1;
    2.  I:= -2;
    3.  I:= (I+2)*J;
```

The first line is read: "I is assigned the value 1", or for short: "I becomes 1".

Multiple assignments are indicated by giving a list of location denotations separated by commas on the left hand side.

Example:

```
    4.  I,J,K := 0;
```

An assignment statement is terminated by a semicolon.

2.1.1. Summary

An assignment statement consists of one or more locations on the left hand side of the assignment symbol and an expression on the right hand side.

An expression consists of one or more operands connected by dyadic operators. An operand is a value, a location, a monadic operation, call of a built_in routine or an expression enclosed in parentheses. A built_in routine call consists of the routine name followed by a list of expressions in parentheses. Using the syntax notation, the above may be stated as:

```
<assignment statement>::=
    <location> {,<location>}* := <expression>;

<expression> ::=
    <operand> {<dyadic operator> <operand>}*
<operand> ::=
    <value>
   |<location>
   |<monadic operator><operand>
   |<built in routine call>
   |(<expression>)
<value>::=
    <literal>
<location>::=
    <name>
<built in routine call>::=
    <routine name> (<expression> {,<expression>}*)
```

2.2. Data Objects and Modes

Data objects are the entities manipulated by the actions of a program. A data object is either a vallue or a location. A location is to be considered as an abstract container where values can be stored. A location should not be confused with physical memory units, although it will occupy one or more such units.

Data objects have a modes attached to them. The mode is a set of properties common to a set of data objects. The mode of an object defines the set of values which the object may assume, the access method if the object is a location, and the valid operations on the values. A number of standard modes are available in the language. Some standard mode denotations are INT, BOOL and CHAR. Others will be introduced later.

Locations always have a unique mode attached to them. For values, however, it is not always possible to attach a unique mode. In that case the value has a class attached. a class is a set of modes constructed from the modes of the locations that may contain the value.

In a program data objects are manifest as denotations. A denotation is for instance a name or a special notation used for basic literals. The denotation of a location is called a location denotation or simply a location. A new location is introduced by a location declaration which allocates a location for its values and attaches a name and a mode to it. Only locations may be assigned to. Values are denoted by value denotations or simply values, e.g. literals. A literal denotes one particular value. In general there is a literal for each value of some class.

Example:

```
    5.  DCL I INT;
    6.  I := I+7;
```

Line 5 is a location declaration statement. I is a location and INT is a mode. In line 6, 7 is a literal.

2.2.1. Discrete modes

Discrete modes are the modes of objects having a finite ordered set of values each of which cannot be divided into subparts. Some of the standard discrete modes are:

INT - The objects are integers of implementation defined size, e.g. -32_768 to 32_767 for a 16 bit computer. An integer literal in its simplest form consists of one or more decimal digits. The underscore symbol may be used to group digits for better readability, otherwise it has no significance. There must be at least one significant digit.

Examples:

```
0
21
1_000_000
```

Binary, octal and hexadecimal integer literals are also provided.

Examples:

```
B'101_111
O'7034
H'FA03
```

BOOL - The boolean or logical mode.
The literals are:

```
FALSE

TRUE
```

CHAR - The mode of one character of the CCITT no.5 alphabet.
A character literal is denoted by enclosing it in single apostrophes:

```
'A'
'B'
' '
'7'
```

An apostrophe character literal is written as ''''. The first and last apostrophes enclose the two middle ones which represent <u>one</u> apostrophe character.

2.2.2. Location Declarations

A location declaration allocates a location, into which a value may be stored, and attaches a name and a mode to it. In its simplest form, a declaration consists of the word DCL (for DeCLare) followed by one or more <u>names</u> of new locations separated by commas, followed by a mode and terminated by a semicolon.

An name must start with a letter possibly followed by a sequence of letters and/or digits and/or underscore characters. Underscore is a significant character in names. Thus the name OFFHOOK is different from OFF_HOOK.

For the sake of readability and documentability, a name should be meaningful and descriptive of the object to which it is attached.

Examples:

```
7.  DCL NO_OF_LINES     INT;
8.  DCL OFF_HOOK, BUSY  BOOL,
9.      ALPHA, BETA     CHAR;
```

In line 7 NO_OF_LINES is declared as the name of a location of INTeger mode. Line 8 declares OFF_HOOK and BUSY as locations of BOOLean mode, and in line 9 ALPHA and BETA of mode CHARacter are declared.

2.2.3. Initialization

Locations may be given initial values when they are declared.

Example:

```
    10. DCL NO_OF_LINES INT := 10_000;
```

This has the same effect as:

```
    11. DCL NO_OF_LINES INT;
    12.     NO_OF_LINES := 10_000;
```

A list of objects of the same mode may be initialized to the same value:

```
    13. DCL X,Y,Z BOOL := FALSE;
```

has the same effect as:

```
    14. DCL X,Y,Z BOOL;
    15.     X,Y,Z := FALSE;
```

Whenever possible, one should initialize within declarations rather than with later assignments. This makes the program easier to read and less error-prone because the locations are guaranteed to have well-defined initial values.

2.2.4. Read-only locations

The programmer sometimes knows that a location is assigned to only once and should later never be modified. This fact may be expressed in the program by prefixing the mode denotation by the word READ.

```
    16. DCL NO_OF_LINES READ INT := 10_000;
```

NO_OF_LINES is now declared to be a read-only INTeger location. It is initialized to 10_000, and any subsequent attempt at assigning to it will produce an error message. Read-only objects must always be initialized.

2.2.5. Summary

```
    <name>::=
        <letter> { <letter> | <digit> | _ }*
    <discrete mode>::=
        [READ] { INT | BOOL | CHAR }
    <integer literal>::=
        <decimal integer literal>
       |<binary integer literal>
       |<octal integer literal>
       |<hexadecimal integer literal>
```

```
<decimal integer literal>::=
    [D'] {<digit>|_}+
<binary integer literal>::=
    B' {0|1|_}+
octal integer literal>::=
    O' {0|1|2|3|4|5|6|7}+
<hexadecimal integer literal>::=
    H' {<hexadecimal digit>|_}+
<hexadecimal digit>::=
    <digit>|A|B|C|D|E|F
<digit>::=
    0|1|2|3|4|5|6|7|8|9
<boolean literal>::=
    FALSE|TRUE
<character literal>::=
    ' CCITT alphabet no. 5 character '
<letter>::=
    A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<declaration statement>::=
    DCL <declaration> {,<declaration>}*;
<declaration>::=
    <name list> <mode> [<initialization>]
initialization>::=
    :=<expression>
```

2.3. Statements and Programs.

A simple CHILL program consists of the word MODULE followed by a sequence of statements, followed by END;.

A module is one of several bracketed constructs in the language. The word MODULE constitutes the opening bracket and END the closing bracket. All bracketed constructs may have a name attached to them by adding the name followed by a colon immediately before the opening bracket. Sometimes the use of such names is necessary to be able to refer to the construct, for instance by an exit statement. This name may be repeated after the corresponding closing bracket, for easy matching of opening and closing brackets. Use of such names is recommended, in particular with nested constructs and when the opening and closing brackets are far apart. This may greatly increase the readability and documentability of the program. The statements are terminated by ";".

The statements are classified as follows

- definition statements
- declaration statements
- action statements

The definition and declaration statements introduce new objects into the language. So far we have only met declaration statements, which create new locations. The definition statements do not create locations. Definition statements will be treated later.

Of the action statements we have only treated the assignment statement. Examples of other action statements are the conditional statements and the loop statements, which will be treated shortly.

Definition and declaration statements must precede action statements. Otherwise there is no particular order imposed on the statements.

Example of a simple CHILL program, which not necessarily does anything useful:

```
1.  COMPUTE_ABSOLUTE_VALUE:
2.  MODULE
3.      DCL I INT:=1,
4.          J INT:=I*2;
5.      I:=J*3;
6.      J:=I/4;
7.      I:=ABS(J-I);
8.  END COMPUTE_ABSOLUTE_VALUE;
```

In line 3, an INTeger object I is declared and initialized to the value 1. The declaration of integers is continued in line 4, an INTeger location J is declared and given the initial value I*2. The current value of I is 1 so I*2 will be 2.

In line 5, the value of J is 2, so the value of J*3 will be 6, which is assigned to I. The operation I/4 on line 6 now produces the value 1 which is assigned to J. Integer division yields the truncated integral part of the quotient.

In line 7, I is 6, J is 1, J-I is -5. ABS is a function or <u>built-in routine</u>. ABS delivers the absolute value of its argument. Thus ABS(J-I) is 5 which is assigned to I.

2.3.1. A Note on Spaces, Linefeeds and Comments

The basic syntactic entities of a CHILL program are called <u>lexical elements</u>. They are divided into three groups:

- names, e.g. I, X, CALL_COUNT, END
- basic integer, character and string literals, e.g. 125, 'A', 'ABCD'
- special symbols, e.g. ";", "+", ":=", "/*"

Lexical elements are terminated by the first character not allowed within the element. The space character is convenient as a terminator for most lexical elements, since it may not appear inside any of them. Exceptions to this rule are character literals and character string literals within which spaces are part of the literal.

Sometimes at least one space is required between elements when confusion might otherwise occur. This is for instance the case between names and between names and basic literals, e.g. MODULEDCL is taken as one name rather than as MODULE DCL which is the start of a module with a declaration statement.

Any number of spaces may appear between lexical elements. Linefeeds and comments have the same separating effect as (one or more) spaces. They may never occur within lexical elements. Comments are enclosed within "/*" and "*/" and may contain any sequence of characters and lineshifts.

Apart from the rules above, CHILL programs may be written in fairly free style, e.g. one statement may extend over several lines.

The proper use of spaces, linefeeds and comments is very important to the readability of a program.

Examples:

```
MODULE AA(I-J)(-I):=-(I+AA(I)(J));END; /* THIS IS VERY COMPACT*/

MODULE

        AA ( I - J )              /* THIS IS THE */
          ( - I ) := - ( I        /* SAME CODE */
          + AA ( I ) ( J ) ) ;    /* VERY EXPANDED */
                     END;

MODULE                            /* BUT THE IMPORTANT */
  AA(I-J)(-I):= -(I+AA(I)(J));    /* THING IS TO MAKE  */
END;                              /* IT READABLE, LIKE */
                                  /* THIS FOR INSTANCE */
```

2.3.2. Input and Output

No standard input and output routines are defined in CHILL. Such routines may be written in CHILL itself. For the sake of writing examples in this book, however, a set of input/output routines are assumed to exist.

Table 2.1. Input/output routines.

routine name	parameter class	result class	action
ININT	none	INT	read one integer value
INBOOL	none	BOOL	read one boolean value
INCHAR	none	CHAR	read one character value
OUTINT	INT	none	write one integer value
OUTBOOL	BOOL	none	write one boolean value
OUTCHAR	CHAR	none	write one character value

2.3.3. Summary

```
<program>::=
    [<name> :] MODULE
        <module body>
    END;
<module body>::=
    {<data statement>}*
    {<action statement>}*
<data statement>::=
    <definition statement>
   |<declaration statement>
```

2.4. Operators and built-in routines

We now continue the presentation of expressions with a closer look at some important operators and built-in routines. Built-in routines are similar to operators except that the operands are written within parentheses following the built-in routine name.

Below are shown a number of examples of expressions using operators and built-in routines. For brevity, only the expressions are shown. In actual programs, the expressions will occur as parts of statements.

Table 2.1. Arithmetic operators

operator	left hand operand class	right hand operand class	priority	result class	action
+	INT	INT	4	INT	Integer addition
-	INT	INT	4	INT	Integer subtraction
-		INT	6	INT	Integer negation
*	INT	INT	5	INT	Integer multiplication
/	INT	INT	5	INT	Integer division
MOD	INT	INT	5	INT	Remainder of integer division

Integer division delivers the truncated integral part of the quotient. There is no rounding. The modulo operator MOD is defined by X MOD Y = X - (X/Y)*Y.

Operators having the highest priority value are evaluated first.

Table 2.2. Relational operators

operator	left hand operand class	right hand operand class	priority	result class	action
=	any	same as left	3	BOOL	Test for equal
/=	any	same as left	3	BOOL	Test for unequal
>	discrete	same as left	3	BOOL	Test for greater
>=	discrete	same as left	3	BOOL	Test for greater or equal
<=	discrete	same as left	3	BOOL	Test for less or equal
<	discrete	same as left	3	BOOL	Test for less

Examples:

Assume the declarations:

```
DCL I INT:=1,
    J INT:=2,
    C CHAR:='Z';
```

Then the following are valid expressions:

```
I=J                  /* 1=2, RESULT: FALSE                 */
C /= 'A'             /* RESULT: TRUE                       */
I+J >= J             /* RESULT: TRUE                       */
I < (J-1)            /* (J-1) IS 1, RESULT: FALSE          */
0 <= (I/2)           /* (I/2) IS 0, RESULT: TRUE           */
(I-2)<0 = (-J<0)     /* (-1<0)=(-2<0), WHICH IS            */
                     /* TRUE=TRUE, RESULT: TRUE            */
```

Table 2.3. Boolean operators

operand	left hand operand class	right hand operand class	prio-rity	result class	action
AND	BOOL	BOOL	2	BOOL	TRUE if both left and right operands are TRUE
OR	BOOL	BOOL	1	BOOL	TRUE if left or right or both operands are TRUE
XOR	BOOL	BOOL	1	BOOL	TRUE if either left or right operand is TRUE and the other is FALSE
NOT		BOOL	6	BOOL	TRUE if operand is FALSE and vice versa.

Examples:

Assume the declarations:

```
DCL I INT:=1,
    J INT:=2,
    B BOOL:=FALSE;
```

Then the following are valid expressions:

```
I<0 OR B                  /* FALSE OR FALSE,                    */
                          /* RESULT: FALSE                      */
I<0 AND J>0               /* FALSE AND TRUE, RESULT: FALSE      */
I<0 OR J>0                /* FALSE OR TRUE, RESULT: TRUE        */
I>0 XOR J>0               /* TRUE XOR TRUE, RESULT: FALSE       */
NOT (I<0)                 /* RESULT: TRUE                       */
NOT (I<0) AND J>0         /* TRUE AND TRUE, RESULT: TRUE        */
I<0 OR.NOT (J>0)          /* FALSE OR FALSE, RESULT: FALSE      */
```

Note that we have used parentheses in "NOT (J>0)". Expressions within parentheses are evaluated prior to the surroundings. If we wrote "NOT J>0", then NOT has higher priority than ">" and would be evaluated first. But then the operand of NOT would be J which is of mode INTeger. There is no NOT operator working on INTegers so an error message will be given.

Table 2.4. Built-in routines

routine name	parameter class	result class	action
ABS	INT	INT	Delivers absolute value of the supplied parameter object
PRED	discrete	Same as parameter mode	PRED(X) delivers, the largest value smaller than X
SUCC	discrete	Same as parameter mode	SUCC(X) delivers the smallest value greater than X
NUM	discrete	INT	Delivers the numerical value of the argument

Examples:

Assume the declarations:

```
DCL I INT := -1,
    C CHAR := 'A';
```

Then these are valid expressions:

```
ABS(I)           /* RESULT : 1  */
PRED(6)          /* RESULT : 5  */
SUCC(C)          /* RESULT: 'B' */
```

2.5. Composite Data Objects

Composite data objects may be built by aggregating other ones.

2.5.1. Arrays

One way of building composite data objects is by means of arrays. An array is composed of objects of the same mode. Each object is called an element of the array. For instance:

```
DCL CALL_COUNT ARRAY(1:5) INT;
```

This statement declares CALL_COUNT to be the name of an array location. The array consists of 5 INTeger elements, with indices (subscripts) 1 to 5. When declaring an array, its size must be known i.e. the index set is usually given by literals or literal expressions. The mode of the array is said to be "ARRAY (1:5) INT".

Elements of CALL_COUNT may be used like any other INTeger location, e.g.:

```
CALL_COUNT(I) := CALL_COUNT(J)+1;
```

CALL_COUNT(I) and CALL_COUNT(J) are array elements of mode INTeger. The operation of accessing an element of an array by means of an index is called indexing. In the above example, the right hand side is evaluated by first doing the indexing for CALL_COUNT(J) and then applying the "+". The left hand side is evaluated by doing the indexing for CALL_COUNT(I). Then the assignment is performed. Which side of the assignment is evaluated first is unspecified in CHILL. Thus one may not rely upon the sides being evaluated in a specific order. The index may, in this case, be any expression yielding an INTeger in the proper value range.

An array operand may in principle be used like any other operand. A value may be assigned to it and arrays may be compared for equality and unequality.

Literals for arrays are called literal tuples. A tuple is just a list of values enclosed in tuple brackets, one value for each element of the array. The tuple brackets are (: and :). Alternatively the square brackets [and] may be used.

For instance, assigning an array literal to an array location may be expressed as:

```
CALL_COUNT := (: 3, 4, 0, 1, 2 :);
```

After this assignment, CALL_COUNT(1) has the value 3, CALL_COUNT(2) has the value 4 and so on.

In comparing arrays, the comparison is done element by element. Both array operands must have the same length.

Example:

```
DCL ERROR BOOL := CALL_COUNT = (: 0,0,0,0,0 :);
```

Here a BOOLean object ERROR is declared and initialized to TRUE if all elements of CALL_COUNT are zero. Otherwise ERROR is set to FALSE.

An array mode, like

```
ARRAY(1:5)INT
```

is a perfectly ordinary mode, and may be used wherever for instance a discrete mode is permitted, provided the resulting construct is meaningful. We may for instance construct arrays of arrays:

```
ARRAY(1:3) ARRAY(1:5) INT
```

This is an array of 3 "ARRAY(1:5)INT" elements, which is a two-dimensional array.

An array of an array is again a perfectly ordinary mode, and we may for instance use it in a *declaration*:

```
DCL MATRIX ARRAY(1:3) ARRAY(1:5) INT;
```

This declares MATRIX to be an object of 3 groups of 5 INTegers each, for a total of 15 INTegers.

MATRIX(3) is then the third group of 5 INTegers. The mode of MATRIX(3) is "ARRAY(1:5)INT". We may access elements of MATRIX(3) like we accessed elements of CALL_COUNT above:

```
MATRIX(3)(5)
```

This is the fifth element in the third group of INTegers in MATRIX. The mode of this element is INTeger. MATRIX may again be used in the normal fashion:

```
MATRIX(I+1)(I-J):= I + MATRIX(I)(J);
MATRIX(I):= (: 3, 4, 5, 1, 2 :);
MATRIX:=    (:(:  5,  4,  3,  4,  5 :),
              (:-10, -5,  0,  5, 10 :),
              (: -8, 16, 24, 32, 41 :) :);
```

2.5.2. Structures

Another way of building composite data objects is by means of structures. A structure is composed of objects of possibly different modes. Each component object is called a field of the structure.

Example:

```
DCL CALL_RECORD STRUCT( JUNCTION_NO,
                        ANSWER_TIME INT,
                        STANDARD_RATE BOOL);
```

This statement declares CALL_RECORD to be the name of a structure location. The structure consists of the fields JUNCTION_NO and ANSWER_TIME, both of mode INTeger and the field STANDARD_RATE of mode BOOLean. The mode of the structure is "STRUCT(JUNCTION_NO, ANSWER_TIME INT, STANDARD_RATE BOOL)".

Individual fields of a structure can be used like any other object of the same mode. The operation of accessing individual fields of a structure is called selection. Selection is done by writing the structure name followed by a dot followed by the field name.

Example:

```
CALL_RECORD.ANSWER_TIME:=TIME+5;
CALL_RECORD.STANDARD_RATE:=TRUE;
```

Structure operands as a whole may be compared for equality and unequality and assigned to. Tuples are used to make literals for structures as well as for arrays.

For instance assigning a structure literal to a structure location may be expressed as:

```
CALL_RECORD := (: 346, 12, FALSE :);
```

After this assignment, CALL_RECORD.JUNCTION_NO has the value 346, CALL_RECORD.ANSWER_TIME has the value 12 and CALL_RECORD.STANDARD_RATE has the value FALSE.

A structure mode may be used in constructing even more complex modes. One may for instance make structures of structures, structures of arrays and arrays of structures.

Example:

```
DCL X STRUCT(A ARRAY(1:3) INT,
             B STRUCT(C,
                      D BOOL)),

    Y ARRAY(1:5) STRUCT( E INT
                         F BOOL,
                         G CHAR);
```

The first declaration above declares a structure X being composed of two fields A and B. A is an array with 3 INTeger elements and B is a structure with two BOOLean fields C and D. The second declaration declares an array Y having 5 elements. Each element is a structure having the fields E, F and G of modes INTeger, BOOLean and CHARacter respectively.

By combining indexing and selection we may get at components of these aggregates:

X.A	is an array
X.A(3)	is the third element of that array, the mode is INTeger
X.B	is a structure
X.B.D	is a BOOLean
Y(2)	is a structure
Y(2).G	is a CHARacter

2.5.3. Read-only composite locations

A composite location as a whole or individual fields of a structure may be protected as read-only.

```
DCL X READ ARRAY (1:3) INT := (: 5, 9, 7 :),
    Y READ STRUCT (B BOOL, C CHAR) := (: TRUE, 'A' :),
    Z STRUCT (D BOOL, E READ CHAR) := (: FALSE, 'B' :);
```

X and Y are completely read-only, both the locations as a whole and their individual components. In Z only the field E is protected, i.e. read-only, also disallowing assignment to the structure as a whole. Thus the following assignment is allowed:

```
    Z.D := Y.B;
```

But all of the following are wrong:

```
X:=(: 1, 2, 3 :);        /* ERROR! */
X(2):=5;                 /* ERROR! */
Y:=Z;                    /* ERROR! */
Y.C:=Z.E;                /* ERROR! */
Z.E:='B';                /* ERROR! */
Z:=(: TRUE, 'C' :);      /* ERROR! */
```

To increase program reliability the read-only protection should be used whenever possible.

2.5.4. Summary

```
<array mode>::=
     [READ] ARRAY(<index set>{,<index set}*)<element mode>
<array literal>::=
     <literal tuple>
<array element>::=
     <array object> (<expression list>)
<structure mode>::=
     [READ] STRUCT(<fields> {, <fields>})
<fields>::=
     <name list> <mode>
<structure literal>::=
     <literal tuple>
<structure field>::=
     <structure object> . <field name>
```

2.6. Conditional statements

Conditional statements are used when parts of a program are to be executed or not depending on certain conditions in the program.

2.6.1. Two-way branch, if statements

This construct describes a two-way branch and allows conditional execution of program parts. An example illustrates this:

```
1.  IF I>0
2.  THEN
3.       OUTCHAR('F');
4.       I-:=1;
5.  FI;
```

The statement is bracketed by the words IF and FI. The word THEN precedes the alternative to be executed when the condition is true.

In line 1, we test whether I is greater than zero. If it is, the actions between THEN and FI are executed, that is lines 3 and 4. OUTCHAR('F') is the call of a procedure to write the character 'F' on some output device. I-:=1 decrements I by one.

There is also a possibility to execute some alternative when the condition is false.

```
 6.  IF I>0
 7.  THEN
 8.       OUTCHAR('F');
 9.       I-:=1;
10.  ELSE
11.       OUTCHAR('E');
12.       I:=0;
13.  FI;
```

Here the actions between THEN and ELSE, that is lines 8 and 9 are executed if I is greater than zero. If I is less than or equal to zero, the actions between ELSE and FI are executed, that is lines 11 and 12.

The text between THEN and FI, or between THEN and ELSE and between ELSE and FI, may in general be any list of action statements. This implies, for instance, that another if statement may follow immediately after THEN or after ELSE. Such an inner if statement is said to be NESTED inside the outer one.

2.6.2. Complicated conditions

Using relational and boolean operators and nested if statements, we can express even the most complicated conditions. However, for the sake of readability and understandability care should be taken to structure the description properly.

In particular deep nesting should be avoided. One tool to that end is the ELSIF construct shown in the example below. In some situations it is more appropriate to use the multiple branch statement (case statement) instead of the if statement. See next section.

Example, digit analysis:

```
14.    DIGIT := INDIGIT();

15.    DIGITS:
16.    IF DIGIT >= 0 AND DIGIT < 10
17.    THEN
18.        VALUE := VALUE*10+DIGIT;
19.    ELSIF DIGIT = -1
20.    THEN
```

```
21.        NEG := TRUE;
22.    ELSE
23.        FIN := NOT FIRST;
24.    FI DIGITS;
```

In line 14 we call the parameterless procedure INDIGIT. Note that a pair of parentheses must always be present in a procedure call, even when the procedure has no parameters. INDIGIT reads the next character. If it is a digit, it returns the value of the digit. If it reads a minus sign INDIGIT returns the value -1. If there are no more digits, some other value is returned. The value returned by INDIGIT is assigned to DIGIT.

In line 16 we test whether this value is greater than or equal to zero and less than 10. If it is, line 18 is executed and the program continues after line 24.

If DIGIT is less than zero or greater than or equal to 10, then the ELSIF part is executed at line 19. This line is an additional test on DIGIT to see if a minus sign has been encountered or if the last digit has been found. In the first case line 21 is executed, otherwise line 23. In both cases, the program is resumed after line 24.

Note that the conditions on line 16 and 19 are expressions, not statements and cannot be terminated by semicolons.

2.6.3. Multiple-way branch, case statements

Often a set of actions is to be selected on the basis of the value of one discrete object. Then the use of a case statement will normally be better than the use of multiple if statements. The program becomes more readable and often more efficient. Going back to our example of digit analysis of the previous section, we may rewrite it as:

```
1.     DIGIT := INDIGIT();
2.     CASE DIGIT OF
3.         (0:9):     VALUE:=VALUE*10+DIGIT;
4.         (-1 ):     NEG:=TRUE;
5.         ELSE       FIN:=NOT FIRST;
6.     ESAC;
```

In line 1 the next character is input. Line 2 expresses a three way branch depending on the value of DIGIT. The parenthesised construct identifying each alternative is called a case label. The name DIGIT is called the case selector. If the value is in the range from 0 to 9, line 3 is executed. If it is -1 then line 4 is executed, else line 5 is executed. In all cases, execution is resumed after line 6. The case statement starts with the word CASE and is terminated with ESAC (CASE spelled backwards). The ELSE alternative is taken when the case selector does not match any of the explicitly stated alternatives. The ELSE part is optional. If it is not present and there is no match for the case selector, there is an error. There must be a unique alternative for each value of the case selector. Overlapping case labels are not allowed.

As another example consider the analysis of a stream of characters. The value of an integer object is to be set according to the class of the input character.

```
 7.    CH:=INCHAR();                                 /* INPUT CHARACTER  */
 8.    CASE CH OF
 9.        ('A' : 'Z'):       I:=1;                  /* ALPHABETIC       */
10.        ('0' : '9'):       I:=2;                  /* NUMERIC          */
11.        (';', ',', '.'): I:=3;                    /* SPECIAL SYMBOL   */
12.        ELSE               I:=4;                  /* ALL OTHERS       */
13.    ESAC;
```

2.6.3.1. The decision table case statement

Complicated conditions will often be much easier to comprehend when expressed in tabular form. Such a table is called a decision table.

```
14.    MODULE
15.        DCL    I READ INT:=ININT(),
16.               C READ CHAR:=INCHAR(),
17.               B READ BOOL:=INBOOL(),
18.               X INT;
19.        CASE   I,          C,            B             OF
20.               ( 1),       ( 'A'),       (TRUE):       X:=1;
21.               (2:5),      ('D':'F'),    (FALSE):      X:=2;
22.               (ELSE),     ('G':'Z'),    ( * ):        X:=3;
23.               ELSE                                    X:=4;
24.        ESAC;
25.    END;
```

In lines 15 to 17 three objects I, C and B are declared and initialized.

Depending on some combinations of values of these three objects, X is to be assigned various values. Line 19 opens the case statement, stating the objects on whose values the selection is dependent. Then in lines 20 to 23 is a list of alternatives to match these values. Line 20 says that if I is 1 and C is 'A' and B is TRUE, then X becomes 1. Line 21 says that if I is in the range from 2 to 5 and C is in the range 'D' to 'F' and B is FALSE, then X becomes 2. Line 22 says that if I is something else than the explicit alternatives of the first column and C is in the range 'G' to 'Z' and irrespective of the value of B, then X becomes 3. The asterisk "*" is the don't care value. If none of these alternatives match, the ELSE alternative at line 23 is taken and X becomes 4.

2.6.4. Summary

```
<if statement>::=
    [<name> :]IF <boolean expression>
          THEN<action statement list>
          {ELSIF <boolean expression> THEN<action statement list>}*
          [ELSE <action statement list>]
          FI [<name>];
<case statement>::=
    [<name> :] CASE <case selector list> OF
            {<case alternative>}*
            [ELSE <action statement list>]
            ESAC [<name>];
<case selector list> ::=
    <case selector> {, <case selector>}*
<case selector> ::=
    <discrete expression>
<case alternative> ::=
    <case label specification> {, <case label specification>}
    <action statement list>
<case label specification>::=
    <case label list>
<case label list> ::=
    (<case label> {, <case label>}*
    |(ELSE)
<case label> ::=
    <discrete literal expression>
    |<literal range>
    |<discrete mode name>
```

2.7. The loop statement

This construct allows repeated execution of program parts. The program part to be repeated may be any list of action statements. These statements are called the body of the loop statement. Repetition is indicated by enclosing the body within the brackets DO and OD. In addition there is a construct for controlling the number of times the loop is executed.

Depending on the kind of loop control, there are two main forms of the loop statement. One is the while form, where a boolean expression indicates when to terminate the loop. The other is the for form where an object, called the loop counter, is stepped through a range of values, such that the loop body is executed once with each value.

2.7.1. The while form

This consists of a while control part in addition to DO - OD. The while control consists of WHILE followed by a boolean expression. The while control is placed after DO.

Example:

```
1.    DO WHILE ABS(I)<100;
2.        I:=ININT();
3.        I:=I*2;
4.    OD;
```

In line 1 we test whether the absolute value of I is less than 100. If it is not, the do statement is terminated and the program continues after line 4. If ABS(I) is less than 100, then the body is executed, that is lines 2 and 3. After executing the body, we repeat from line 1. The condition is again evaluated and tested to see whether the body should be executed once more. Note that if the condition is false initially, the body will not be executed at all.

2.7.2. The for form

The for control part consists of FOR followed by an iteration specification.

2.7.2.1. The indefinite loop

The very simplest for control is EVER which describes an indefinite loop.

Example:

```
5.    DO FOR EVER;
6.        I+:=1;
7.        WAIT(5);
8.    OD;
```

The loop body is repeated indefinitely. In this case a counter I is incremented every 5 seconds. WAIT is assumed to be a procedure which delays the program for a given number of seconds. The loop may be terminated by some condition independent of the loop statement. This may, for instance, be an exit statement.

2.7.2.2. Loops with loop counters

Normally FOR is followed by a loop counter followed by an in part indicating the values to be stepped through. The simplest kind of in part consists of IN followed by a discrete mode, giving the smallest and the largest values of the loop counter. The loop counter should not be declared elsewhere, as the for control automatically declares it within the do statement. The class of the loop counter is determined by the in part. The counter may not be assigned to within the loop body (it is a value denotation) and does not exist outside the do statement.

Example:

```
9.     DO FOR I IN INT(1:100);
10.        SUM+:=I;
11.    OD;
```

This loop computes the sum of all integers from 1 to 100, assuming that SUM is 0 prior to entering the loop. Line 10 is the loop body that is repetitively executed. Line 9 declares the loop counter to be of some integer class having the value range from 1 to 100. It will be 1 the first time line 10 is executed, 2 the second time and so on. Thus the loop will be executed 100 times.

The in part may contain DOWN before IN, and the for control may be combined with the while control.

Example:

```
12.    DO FOR CH DOWN IN CHAR('A':'Z')
           WHILE CH /= 'F';
13.        OUTCHAR(CH);
14.    OD;
```

This loop will print the characters from 'Z' to 'G' in reverse alphabetical order.

Using this particular form of the in part, the loop counter may only be incremented or decremented by one unit. In practice, this is not felt as a restriction, mainly because of a particular form of the in part for scanning through arrays. See next section.

If there is a need, however, for using other step sizes the language offers the following construct:

Example:

```
15.    DO FOR I := 1 BY 3 TO 100;
16.        SUM +:=I;
17.    OD;
```

2.7.2.3. Scanning arrays

An in part may consist of IN followed by an array location The loop counter is now a location which will represent each element of the array, in turn.

Example:

```
18.   DCL A READ ARRAY (1:100) INT := INARRAY(),
19.       SUM1, SUM2 INT := 0;
20.   DO FOR ELEMENT IN A;
21.       SUM1 +:= ELEMENT;
22.       SUM2 +:= ELEMENT*ELEMENT;
23.   OD;
```

The loop in lines 20 to 23 will compute two sums. SUM1 becomes the sum of all elements of A and SUM2 becomes the sum of all squares of the elements.

The utilization of this construct for scanning arrays, rather than using explicit indexing, is very important for the sake of program reliability and efficiency. It is guaranteed that ELEMENT will scan through exactly all the elements of A, not one more nor one less. Also the size of A may be changed without changing the for control.

We may also use DOWN in this kind of in part to scan the array in reverse order.

2.7.3. Summary

```
<do statement> ::=
    [<name> :]
    DO [<control part>]
    <action statement list>
    OD [<name>];
<control part> ::=
    <for control> [<while control>]
    |<while control>
<for control> ::=
    FOR { <interation> { , <iteration> }* | EVER }
<iteration> ::=
    <value enumeration>
    |<location enumeration>
<value enumeration> ::=
    <step enumeration>
    |<range enumeration>
<step enumeration> ::=
    <loop counter name> ::= <start value> [<step>][DOWN]<end value>
<step> ::=
    BY <integer expression>
<range enumeration> ::=
    <loop counter> [DOWN] IN <discrete mode>
```

```
<location enumeration> ::=
    <loop counter name> [DOWN] IN <array location>
```

2.8. The exit statement

Sometimes there is a need to terminate the execution of a bracketed action statement before reaching the closing bracket. This may for instance be the case in error situations. For that purpose, the exit statement may be used. The statement consists of the word EXIT followed by the name of the bracketed construct to be terminated.

Example:

```
1.    CONSUMER:
2.    DO FOR EVER;
3.        EMPTY_BUFFER(STATUS);
4.        IF STATUS = ERROR
5.        THEN
6.            EXIT CONSUMER;
7.        FI;
8.        RETURN_BUFFER();
9.    OD CONSUMER;
```

This is an indefinite loop which receives filled buffers from some producer and empties them. If error status is returned, the loop is terminated by the exit statement of line 6. Otherwise the empty buffer is returned. After performing the exit statement, execution is resumed after line 9.

2.8.1. Summary

```
<exit statement>::=
    EXIT <bracketed construct name>;
```

2.9. Procedures

The procedure facility is a fundamental abstraction and structuring tool in CHILL programs.

A procedure is a closed subprogram which is entered only by means of a procedure call. When the procedure actions have been performed, the program is resumed immediately following the call.

2.9.1. Procedure calls

We have seen examples of procedure calls in the preceeding text. A call consists of the procedure name followed by left parenthesis, followed by a list of zero or more parameters followed by right parenthesis. Note that the set of parentheses must be present even if there are no parameters.

Example:

```
1.    ANALYZE_DIGIT(CURRENT_DIGIT);
```

The parameters given in the call are termed the actual parameters. An actual parameter may be any expression yielding a value of the appropriate class. Each actual parameter is evaluated once and the yielded value is made available to the statements inside the procedure.

A procedure may yield a value as result. In this case the procedure call is a value denotation and may be used as an operand in an expression.

Example:

```
2.    X:= MAX(X,Y)*2;
```

X and Y are assumed to be integers and the procedure MAX returns the value of the larger of the two.

2.9.2. Procedure definitions

The procedure definition starts with the procedure name followed by colon followed by the word PROC followed by formal parameter specification and result mode specification terminated by a semicolon. The definition is terminated with END. Inside, the procedure consists of a statement list called the procedure body.

The formal parameter specification is a list of parameter names and associated mode denotations enclosed in parentheses. If there are no parameters, an empty pair of parentheses is written. If the procedure delivers no result, the result mode specification is omitted. When present, the result mode must be enclosed in parentheses.

A very simple procedure, with no parameters and no result value, may for instance be defined by

```
    CHECK:
 3. PROC();
 4.       IF I<0
 5.       THEN
 6.            OUTTEXT ('ERROR: I WAS LESS THAN 0');
 7.            I:=0;
 8.       FI;
 9.    END CHECK;
```

This procedure should be called by writing

```
10.    CHECK();
```

Each time the call is performed, the procedure body will be executed, that is lines 4 to 8. In lines 4 and 7, I is an object which in this case is assumed to be declared outside the procedure.

The procedure (i.e. the procedure body) checks that the value of I is less than 0. If it is, an error message is printed and I is set to zero.

Consider another example, a procedure with two parameters and a result:

```
       MAX:
11.    PROC(I, J READ INT) (INT);
12.         IF I > J
13.         THEN
14.              RETURN I;
15.         ELSE
16.              RETURN J;
17.         FI;
18.    END MAX;
```

This defines the procedure MAX which yields as a result the value of the larger parameter. The "(I, J READ INT)" is the formal parameter specification. There are two formal parameters I and J both of mode READ only INTeger. The parameters are specified as READ only because we know that the procedure should not change them. We may, however, still supply non-READ objects as actual parameters. The names of the formal parameters are visible, i.e. usable, only inside the procedure body. They are said to be local to the body. "(INT)" is the result mode specification. The value returned is of integer class.

The result value returned by the procedure is indicated by a return statement, like in lines 14 and 16.

To call the procedure, actual parameters must be supplied.

Example:

```
19.    R:=5*MAX(K+L,M)+3;
```

The actual parameters are K+L and M.

2.9.3. Parameter passing

The process of substituting the actual parameters for the formal ones is called parameter passing. In its simplest form, parameter passing works like declarations with initializations.

The call on MAX in line 19 works as if the following declarations were performed at the place of the formal parameter specification:

```
DCL I INT:= K+L,
    J INT:= M;
```

I and J behave like normal INTeger objects inside the procedure body. This means that the values of the actual parameters have been copied into objects local to the procedure. Such a parameter mechanism is called "pass by value". This method is very efficient for objects of small size, but the copying may become expensive for large objects.

2.9.4. Special passing of composite object parameters

When an actual parameter is a large array or structure, we would not like to produce a complete local copy inside the procedure. This would be far to inefficient. We would rather pass some compact description of the object enabling the procedure to access it. In order to achieve this, write LOC behind the mode of the formal parameter.

Example:

```
        MAX_ELEMENT:
20.     PROC (X READ ARRAY(1:100)INT LOC) (INT);
21.         DCL CURRENT_MAX INT := X(1);
22.         DO FOR ELEMENT IN X;
23.             IF ELEMENT > CURRENT_MAX
14.                   THEN
25.                       CURRENT_MAX:=ELEMENT;
26.             FI;
27.         OD;
28.         RETURN CURRENT_MAX;
29.     END MAX_ELEMENT;

30.     DCL B ARRAY (1:100) INT:=INARRAY(),
31.         BMAX INT := MAX_ELEMENT(B);
```

In this example lines 20 to 29 define a procedure MAX_ELEMENT which selects the largest element out of an INTeger array.

In line 20 "READ ARRAY(1:100)INT LOC" specifies X to be an object describing any array of 100 integer elements. We specify X to describe READ only arrays, because we know that the procedure body will never change any element of the actual parameter array. In line 21 an integer object CURRENT_MAX is declared and initialized to the first element of the array described by X. In line 22 the object ELEMENT is implicitly declared and specified to represent in turn all the elements of the array described by X. Lines 23 to 26 set CURRENT_MAX to the largest value found so far. Line 28 specifies that the value yielded by the procedure is the final value of CURRENT_MAX.

Line 30 declares an INTeger array B of 100 elements and reads some input to initialize it. Line 31 declares an INTeger object BMAX and initializes it to the value of the largest element of B by calling the procedure MAX_ELEMENT.

2.9.5. Results of a procedure call

If a procedure is to yield a result, it is recommended that the result is conveyed as the value of the procedure itself, as is the case with the procedures MAX and MAX_ELEMENT above. The result to be returned in this way, is indicated within the procedure body by a return or result statement, see lines 14, 16 and 28.

In addition to communicating with the calling program via a result or via the parameters, as shown in the examples, the procedure body may access global objects, i.e. objects declared outside the procedure definition. The procedure may, for instance, assign a computed value to an object declared outside the procedure to make this value available to others. Such modification of global objects is called a side-effect of the procedure. Side-effects should in general be avoided because they tend to make the program unstructured and difficult to handle.

2.9.6. The return and result statements

When the execution of a procedure reaches the END of the procedure, control is returned to the place of the call. Control may be returned at any earlier point by using the return statement. It is simply written:

```
        RETURN;
```

If the procedure yields a result, the return statement may also indicate the value yielded. If there is a need to indicate the yielded result at a point different from the return point, the result statement can be used.

Example:

```
RETURN X+5;
```

This is equivalent to

```
RESULT X+5;
RETURN;
```

2.9.7. Summary

```
<procedure definition statement>::=
    <name>: <procedure definition> [<name>];
<procedure definition> ::=
    PROC ([<formal parameter list>]) [<result spec>];
    <proc body>
    END
<formal parameter list> ::=
    <formal parameter> {, <formal parameter>}*
<formal parameter> ::=
    <name list> <mode>
<procedure call> ::=
    <procedure name> ([<actual parameter list>])
<actual parameter list> ::=
    <value> { , <value> }*
<return statement> ::=
    RETURN [<value>];
<result statement>::=
    RESULT <value>;
<result spec> ::=
    (<mode>)
```

3. Data object description

In the previous chapter most of the basic action descriptions of CHILL were presented together with some of the elementary data objects. In this chapter, the object descriptions will be explained more thoroughly.

Describing the properties of objects, forms a complement to the description of actions. This duality removes some possibilities for making errors and permits an extensive checking of the consistency in programs, thus greatly contributing to security and reliability. Also errors which otherwise would be discovered only at run-time may now be caught at earlier stages in the program development process.

3.1. Synonyms

Basic literals for integers, booleans and characters were introduced earlier. They are a fixed part of the language. The synonym concept allows the programmer to define new literals. These user defined literals must always be in the form of names. A synonym definition consists of the word SYN followed by a name followed by the definition sign "=" followed by an expression whose operands are other literals. The name becomes a new literal for the value yielded by the literal expression.

Example:

```
    SYN BUFFER_SIZE = 80,
        NO_OF_BUFFERS = 10,
         UFFER_POOL_SIZE = BUFFER_SIZE * NO_OF_BUFFERS;
    DCL BUFFER_POOL ARRAY(1:BUFFER_POOL_SIZE) INT;
```

The use of synonyms is important for program parameterization, and for providing meaningful names to "magic" literals. This makes programs more readable and easy to change.

3.1.1. Summary

```
<synonym definition statement> ::=
        SYN <synonym definition> {,<synonym definition>}*;
<synonym definition> ::=
        <name list> [<mode>] = <literal value>
```

3.2. Modes and classes

A mode is a set of properties common to a set of objects. We have seen some already, in particular integer, boolean, character, array and structure modes. All program constructs delivering data objects, have an attached mode or class. They determine, for instance, the set of possible values the object may assume and which operations may be performed on the data object. At least some predefined or built-in operations are applicable to the objects of each mode. To integer objects, for instance, we may apply the usual "+", "*" etc. operators. To all objects of any array mode, we can apply the indexing operation, that is to pick out the element selected by an index.

Part of the purpose of modes is to ensure that not all operations may be applied to any object. Modes allow the objects of a program to be structured into a number of incompatible groups, each for their own purpose, each described by its own mode. This tends to increase the clarity of the programs and also enables a significant number of inconsistencies to be detected, when they result in mode errors.

Modes also reduce the error rate by simplifying the actions. Modes indirectly indicate how much machine storage is needed for each object and how the components of a data structure are related to each other. This enables the compiler to decide a number of details of data object layout, and how to access these objects. Thus the programmer is relieved of much error prone work on data allocation and accessing, e.g. constant table generation, indirect addressing, shifting and masking etc. Defining and denoting modes will be found to form a significant part of most useful CHILL programs. In fact, the beginner to CHILL programming may be unpleasantly surprised by the amount of declarations and definitions that must be written before "the real" program begins, that is the algorithmic or executable parts. We are in essence concentrating into the mode definition information that would otherwise be spread out over the entire program, thus making it easier to comprehend and modify.

3.2.1. Mode definitions

Modes appear in programs in the form of mode denotations or just modes for short. Some are standard in the language like for instance the modes INT, BOOL and ARRAY(1:3)INT which we have seen before.

New modes may be defined in terms of already defined ones by means of mode definitions. There are two kinds of mode definition statements, the new mode definition and the synonym mode definition.

3.2.1.1. New modes

A new mode definition statement consists of the word NEWMODE followed by a name, a definition sign and a mode. The name becomes a new denotation for a distinct mode having all the properties of the defining mode.

Example:

```
    1.  NEWMODE MATRICES = ARRAY(1:3)ARRAY(1:5) INT;
    2.  DCL MATRIX_A MATRICES;
```

In line 1 "ARRAY(1:3)ARRAY(1:5)INT" is a mode. MATRICES is defined to be a new mode for 3 by 5 arrays of INTegers. Line 2 declares MATRIX_A to be a particular object of that mode...

Example:

```
    3.  NEWMODE JUNCTIONS = INT,
    4.          TIMES = INT,

    5.          CALL_RECORDS = STRUCT(JUNCTION_NO JUNCTIONS,
    6.                                ANSWER_TIME TIMES,
    7.                                STANDARD_RATE BOOL);

    8.  DCL CALL_RECORD_AREA ARRAY(1:1000) CALL_RECORDS,
    9.      CURRENT_CALL_RECORD CALL_RECORDS:= CALL_RECORD_AREA(1);

   10.      CURRENT_CALL_RECORD.JUNCTION_NO := 1375;
   11.      CURRENT_CALL_RECORD.ANSWER_TIME := 3;
   12.      CUR  ..T_CALL_RECORD.JUNCTION_NO :=
   13.          CURRENT_CALL_RECORD.ANSWER_TIME; /* MODE ERROR! */
```

Lines 3 and 4 define two new modes JUNCTIONS and TIMES. Although they both have the properties of integers, they are different from each other and from INT.

In line 5 CALL_RECORDS is defined to be the .ode of data objects having the form given by the structure on the right hand side of "=". Line 8 declares an object consisting of 1000 such call records, and line 9 declares one particular call record which is initialized to the first element of CALL_RECORD_AREA. (Presumeably CALL_RECORD_AREA has been properly initialized somewhere).

Lines 10 and 11 assign new values to JUNCTION_NO and ANSWER_TIME. The assignment statement of lines 12 and 13 is, however, in error, because the fields have been given different modes. This is reasonable because junction numbers and answer times are basically different quantities, even though they both have the properties of integers.

It is hoped that these examples show that proper use of mode definitions greatly facilitates consistency checking, improves the ..adability of the program and makes it easier to write and modify. This is particularly true when complicated modes are used several times.

3.2.1.2. Synonym modes

A synonym mode definition statement consists of the word SYNMODE followed by a name, a definition sign and a mode. The defined name becomes another mode denotation for the defining mode. This means that we can substitute the defining mode at every application of the synmode name without changing the semantics of the program.

Example:

```
14.  SYNMODE S_MATRICES = ARRAY(1:3)ARRAY(1:5) INT;
15.  DCL MATRIX_B S_MATRICES,
16.      MATRIX_C ARRAY(1:3)ARRAY(1:5) INT;
17.  MATRIX_B := MATRIX_C;
```

Line 14 defines S_MATRICES to be another mode for 3 by 5 arrays of INTegers. Lines 15 and 16 declares two objects MATRIX_B and MATRIX_C which are of the <u>same</u> mode. Thus the assignment on line 17 is legal.

In chosing between newmodes and synmodes, the most restricted concept, i.e. newmodes, should be used, unless the extra flexibility of synmodes is really needed.

3.2.2. Set modes

Set modes are the modes of objects having a (often small) discrete set of values. A set mode consists of the word SET followed by a list of names which are the literals enumerating all the values of the set. Suppose we want to describe the states of a phone line, a mode for these states might be

```
18. NEWMODE LINE_STATES = SET(ON_HOOK, OFF_HOOK, BUSY, IDLE,
                                                  DISABLED);
```

This statement defines LINE_STATES to be a mode denotation for objects which may assume 5 values as given by the literals ON_HOOK, OFF_HOOK, BUSY, IDLE and DISABLED.

Next we may want to declare an object for keeping the current state of the line:

```
19. DCL CURRENT_STATE LINE_STATES:=ON_HOOK;
```

CURRENT_STATE is initialized to the value ON_HOOK. Objects of mode LINE_STATES can be used like any other objects. We may for instance assign to them:

```
20. CURRENT_STATE:=POLL_LINE_STATE();
```

POLL_LINE_STATE is assumed to be a procedure interrogating the line and delivering a value of mode LINE_STATES. We may then ask whether the line has a particular state:

```
        IF CURRENT_STATE = BUSY
        THEN
            GIVE_BUSY_TONE();
        FI;
```

We may [illegible] make a table for taking proper action for all possible states:

```
        CASE CURRENT_STATE OF
            (ON_HOOK):      /* ACTION STATEMENTS */;
            (OFF_HOOK):     /* ACTION STATEMENTS */;
            (BUSY):         /* ACTION STATEMENTS */;
            (IDLE):         /* ACTION STATEMENTS */;
            (DISABLED):     /* ACTION STATEMENTS */;
        ESAC;
```

Note that no else alternative is provided above. This makes it possible to check that all cases have been considered. If one case is missing, an error message will be produced. For instance if another state is added to the state set, but forgotten in the case statement, this will be noticed.

For the sake of testing, we may want to produce successively all possible states of a line:

```
DO FOR TEST_STATE IN LINE_STATES;
/* ACTION STATEMENTS */
OD;
```

Within this loop the object TEST_STATE will in turn take on the values ON_HOOK, OFF_HOOK etc. Note that if the number of states in the state set is changed, the above statement is still valid.

The built_in routines PRED and SUCC also work on set mode objects, to give the predecessor or successor values respectively.

Examples:

```
CURRENT_STATE := BUSY;
CURRENT_STATE := SUCC(CURRENT_STATE); /* THE NEW VALUE ASSIGNED
                                         TO CURRENT_STATE IS
                                         IDLE */
CURRENT_STATE := SUCC(DISABLED);      /* ERROR! */
CURRENT_STATE := PRED(ON_HOOK);       /* ERROR! */
```

3.2.3. Range modes

Out of any discrete mode, new discrete modes may be made for objects having a range of values being part of the set of values of the parent mode.

Examples:

```
NEWMODE BYTE = INT(0:255),
        SHORTINT = INT(-128:127),
        ALPHABETICS = CHAR('A':'Z'),
        NORMAL_STATES = LINE_STATES(ON_HOOK:IDLE);
```

In general it is recommended to use ranges rather than the parent mode, whenever the full range of values of the parent mode are not needed. In particular this is the case for integers. This will aid concistency checking and on many computers it may improve the space/time efficiency of programs.

3.2.4. Summary

```
<newmode definition statement> ::=
        NEWMODE <mode definition> {,<mode definition>}*;
<synmode definition statement> ::=
        SYNMODE <mode definition> {,<mode definition>}*;
<mode definition> ::=
        <name list> = <defining mode>
<set mode> ::=
        SET ( <set element> {, <set element>} *);
<range mode> ::=
        <discrete mode name> ( <literal range> )
<literal range> ::=
        <lower bound> : <upper bound>
<lower bound> ::=
        <discrete literal expression>
<upper bound> ::=
        <discrete literal expression>
```

3.3. String modes

Strings are a special kind of arrays whose elements are characters or bits. Special operations and special literals are provided to allow convenient handling of these kinds of objects.

3.3.1. Character strings

The mode for character strings, consists of the word CHAR followed by a length enclosed in parentheses. A character string literal is a sequence of characters enclosed in single apostrophes. One apostrophe within a character string literal is written as two apostrophes. The literal may contain a repetition factor in front, enclosed in parentheses.

Example:

```
1.  DCL LINE_BUFFER CHAR(25):=(25)' ';
2.  LINE_BUFFER := 'ERROR MESSAGE TO OPERATOR';
3.  OUTSTRING (LINE_BUFFER);
4.  LINE_BUFFER := 'THE KEY IS ''VOLUME 5''     ';
5.  OUTSTRING(LINE_BUFFER);
```

Line 1 declares a string LINE_BUFFER of 25 characters and initializes it to 25 spaces. Line 2 assigns a string literal to LINE_BUFFER and in line 3 it is printed.

Line 5 will print the text: THE KEY IS 'VOLUME 5'.

3.3.2. Bit strings

The mode for bit strings consists of the word BIT followed by a length in parentheses. The literals are similar to the character string literals, except that there is an indication whether the string literal is in binary, octal or hexadecimal form.

Example:

```
6.  DCL STATUS_WORD BIT(16) := H'3F5A';
7.  NEWMODE BYTE = BIT(8);
8.  DCL B1, B2 BYTE := B'1000_0001';
9.  DCL ADDRESS BIT(18) := O'301_526';
```

Line 6 declares the object STATUS_WORD to consist of 16 bits initialized to the hexadecimal string literal H'3F5A'. In line 7 a new mode BYTE is defined to be the denotation for 8-bit objects. Two byte objects B1, and B2 are declared in line 8 and initialized to a binary string literal. Line 9 declares an object ADDRESS of 18 bits and initializes it to the octal string literal O'301_526'. The underscore within the literal strings is used to group digits for better readability. Apart from that, the underscore has no effect.

Note the difference between bit string literals and binary, octal and hexadecimal integer literals. The latter are not terminated by an apostrophe.

3.3.3. String operations

To access part of a character or bit string the operations indexing and substringing are defined. Indexing consists of the string object followed by an index in parentheses. Substringing consists of the string object followed by an indication of which substring to access. This indication is either in the form of a range or a start position and a length.

Example:

```
10.     DCL OVERFLOW_INDICATOR BIT(1):=STATUS_WORD(0),
11.         CONDITION_CODE BIT(3):=STATUS_WORD(2:4),
12.         SUB_LINE CHAR(15):=LINE_BUFFER(10 UP 15);
```

Strings are indexed from zero up to its length minus one. In line 10 bit no. 0 is indexed from STATUS_WORD. In line 11 bits no. 2, 3 and 4 are sliced and in line 12 characters no. 10, 11, 12 .... 23, 24 are sliced.

In addition to the relational operators, the operators defined on strings are shown in table 3.1.

Table 3.1. String operators

operator	left hand operand class	right hand operand class	priority	result class	action
//	string	string	4	string	Concatenate operands
AND	BIT(n)	BIT(n)	2	BIT(n)	Bitwise AND
OR	BIT(n)	BIT(n)	1	BIT(n)	Bitwise OR
XOR	BIT(n)	BIT(n)	1	BIT(n)	Bitwise XOR
NOT	BIT(n)	BIT(n)	6	BIT(n)	Bitwise NOT

When strings are compared, the comparison is done element by element from low to high indices as long as the relation remains true.

Example:

```
'ABCD' > 'ABCF'
```

This relation yields FALSE because 'D' < 'F'

With concatenation the right operand is appended to the left one producing a new string whose length is the sum of the operand lengths.

Example:

```
13. LINE_BUFFER := 'NO ERROR' // (17)' ' ;
```

Here a new message 'NO ERROR' is assigned to LINE_BUFFER. However, to perform string assignment, the left and right hand sides must have equal length. Because LINE_BUFFER has length 25, we concatenate 17 spaces to the string literal in order to make 25 characters on the right hand side.

3.3.4. Summary

```
<string mode> ::=
        {CHAR|BIT} (<string length>) [VARYING]
<character string literal> ::=
        ' {<non apostrophe character> | <apostrophe>}*'
        | C' {<hexadecimal digit> <hexadecimal digit>}*'
bit_string literal> ::=
        <binary bit_string literal>
        | <octal bit_string literal>
        | <hexadecimal bit_string literal>

<binary bit_string literal> ::=
        B' {0 | 1 | _}*'

<octal bit_string literal> ::=
        O' {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | _}*;

<hexadecimal bit string literal> ::=
        H' {<hexadecimal digit> | _}*'

<string element> ::=
        <string location> (<string index>)

<substring> ::=
<string location>(<left element> :  <right element>)
        |<string location>(<position> UP <length>)
```

3.4. Referencing modes

It is sometimes convenient or even necessary to work with references (addresses or descriptors) to data objects rather than the objects themselves. For this purpose CHILL provides reference and row modes.

3.4.1. Reference modes

Reference modes define values referring to locations of static mode. They may also be empty, i.e. not pointing at any object. The only reference literal is the literal NULL denoting the empty pointer. In order to create other reference values the referencing operator "->" is used. It is a monadic operator taking a location denotation as right operand and delivering the address of that location as result.

A reference mode is qualified by the mode of the locations refered to. This restriction is essential for program security. The mode consists of the word REF followed by the mode of the objects referenced.

Example:

```
     1.      DCL   I INT := 2,
     2.            J INT := 3,
     3.            RI REF INT := ->I;
```

Lines 1 and 2 declare two locations I and J initialized to 2 and 3 respectively. Line 3 declares a location RI of mode "REFerence to INTeger", that is it may contain addresses of INTeger locations. RI is initialized to refer to the location of I. Thus the following picture has been created:

```
  Data names                 Data objects                Data modes
  ----------                 ------------                ----------

                              ------------
      I                    -->:         2 :                  INT
                           !  ------------
                           !
                           !
                           !  ------------
      J                    !  :         3 :                  INT
                           !  ------------
                           !
                           !
                           !  ------------
      RI                   !--:----*      :                  REF INT
                              ------------
```

The location I may now be accessed directly via its name I or indirectly via the reference RI. To indicate indirect access via a reference, the dereferencing or "follow-the-pointer" operator "->" is used. It takes a reference value as left operand and yields the location pointed at.

```
4.      J := RI-> +4;
5.      RI := ->J;
6.      I := I*RI->;
```

In line 4 the right hand side takes the value of the location referenced by RI and adds the value 4. The location referenced by RI is I, thus the value assigned to J is 6. In line 5, RI is changed to refer to J. In line 6 the value of I, i.e. 2, is multiplied by the value of the location referenced by RI i.e. 6. The value assigned to I is 12. Now the picture is:

```
        Data names              Data objects              Data modes
        ----------              ------------              ----------


                                 -----------
            I                    :       12 :                INT
                                 -----------


                                 -----------
            J               ---->:        6 :                INT
                            !    -----------
                            !
                            !
                            !    -----------
            RI              !----:----*     :                REF INT
                                 -----------
```

3.4.2. List handling

Relationships between objects may often be expressed by grouping the objects together in one composite object, i.e. structure or array. Such grouping is usually sufficient for describing simple, static relationships. More complex or dynamic relationships may require the explicit use of references to link the related objects.

The choice of which relationships are to be represented by grouping and which to be represented by explicit linking, will depend on the circumstances. In any case, one will usually find structure objects and references used in concert.

In the following we will consider an example where a doubly linked circular list is to be created.

Example:

```
 7.  CIRCULAR_LIST:
 8.  MODULE
 9.       /* DOUBLY LINKED CIRCULAR LIST */
10.       NEWMODE NODE = STRUCT(PRED, SUC REF NODE, VALUE INT);
11.       DCL HEAD NODE := (: NULL, NULL, 0 :);
12.       DCL POOL ARRAY(1:1000) NODE;
13.       DCL LAST REF NODE := ->HEAD;
14.       DO FOR NEW IN POOL;
15.            NEW.PRED := LAST;
16.            LAST->.SUC := ->NEW;
17.            LAST := ->NEW;
18.            NEW.VALUE := 0;
19.       OD;
20.       HEAD.PRED := LAST;
21.       LAST->.SUC := ->HEAD;
22.  END CIRCULAR_LIST;
```

The module containing the doubly linked list is called CIRCULAR_LIST. Line 10 defines the mode, called NODE, of each list element to be a structure composed of three fields. The two first fields are references of mode REF NODE. There is a reference to the predecessor in the list called PRED and to the successor called SUC. Finally there is a data field called VALUE of mode INT.

Line 11 declares the first element of the list called HEAD and initializes its PRED and SUC to empty, i.e. NULL and VALUE to zero. Line 12 declares an array object of 1000 NODE elements. In line 13 an auxiliary reference LAST is declared and initialized to point to the element HEAD.

Lines 14 to 18 constitute a loop for creating the linked list. NEW will represent in turn all the NODE elements of POOL. The reference LAST will trail one element behind NEW. Line 15 sets the predecessor reference and line 16 the successor reference of each element. Finally LAST is moved on to the next element in line 17. In lines 20 and 21 the circle is closed by tying the last element of POOL to the HEAD.

When this module has been executed, the following picture has been created:

```
         Data names             Data objects         Data modes
         ----------             ------------         ----------
                                        -------
             : HEAD.PRED          !-----:-*   :<-----    REF NODE :
             :                    !  -->:     :     !             :
             :                    !  !  -------     !             :
HEAD         : HEAD.SUC           !  !  :  *--:--!  !    REF NODE : NODE
             :                    !  !  :     :  !  !             :
             :                    !  !  -------  !  !             :
             : HEAD.VALUE         !  !  :  0  :  !  !    INT      :
                                  !  !  :     :  !  !
                                  !  !  -------  !  !
                                  !  !           !  !
                                  !  !           !  !
                                  !  !  -------  !  !
             : POOL(1).PRED       !  !--:--*  :<--   !    REF NOD   :
             :                    !  -->:     :     !             :
             :                    !  !  -------     !             :
    L(1)     : POOL(1).SUC        !  !  :  *--:--!  !    REF NODE : NODE
             :                    !  !  :     :  !  !             :
             :                    !  !  -------  !  !             :
             : POOL(1).VALUE      !  !  :  0  :  !  !    INT      :
                                  !  !  :     :  !  !
                                  !  !  -------  !  !
                                  !  !           !  !
                                  !  !           !  !
                                  !  !  -------  !  !
             : POOL(2).PRED       !  !--:--*  :<--   !    REF NODE :
             :                    !  -->:     :     !             :
             :                    !  !  -------     !             :
POOL(2)      : POOL(2).SUC        !  !  :  *--:--!  !    REF NODE : NODE
             :                    !  !  :     :  !  !             :
             :                    !  !  -------  !  !             :
             : POOL(2).VALUE      !  !  :  0  :  !  !    INT      :
                                  !  !  :     :  !  !
                                  !  !  -------  !  !
                                  !  !           !  !
                                  !  !           !  !
                                  !  !           !  !
                                  !  !  -------  !  !
             : POOL(1000).PRED    !  !--:--*  :<--   !    REF NODE :
             :                    -->-->:     :     !             :
             :                       !  -------     !             :
POOL(1000)   : POOL(1000).SUC        !  :  *--:-----!    REF NODE : NODE
             :                       !  :     :                   :
             :                       !  -------                   :
             : POOL(1000).VALUE      !  :  0  :          INT      :
                                     !  :     :
                                     !  -------
                                     !
                                     !
                                     !
                                     !  -------
LAST                                 !--:--*  :          REF NODE
                                        -------
```

3.4.3. Row modes

A declared array has a fixed size. Sometimes we may want to work on arrays of unknown size, or arbitrary parts of arrays. This fact is described in the program by means of a row object.

A row is a descriptor for arrays or parts of arrays. It contains a pointer to the start of (the part of) the array being described and the number of elements described.

The use of rows facilitates efficient and flexible array processing. It is also very powerful in text processing, and in general where slices, i.e. parts of arrays, are needed.

```
    22.    DCL M ARRAY(1:5)CHAR:=(: 'H', 'I', 'T', 'C', 'H' :),
    23.          R ROW ARRAY(1:5) CHAR:= <-M;
```

Line 22 declares M to be an array of 5 CHARacters. The first element is initialized to 'H', the second to 'I' and so on. Line 23 declares R to be a ROW describing any (sub)array of CHARacters having 5 elements or less. R is initialized to describe all of M. We now have:

```
      Data names                 Data objects      Data modes
      ----------                 ------------      ----------

                                   -------
      :  M(1)  :  R->(1)   :     : 'H' :<--      : CHAR :
      :                            -------   !          :
      :  M(2)  :  R->(2)   :     : 'I' :   !     : CHAR :
      :                            -------   !          :
    M :  M(3)  :  R->(3)   :     : 'T' :   !     : CHAR : ARRAY(1:5)CHAR
      :                            -------   !          :
      :  M(4)  :  R->(4)   :     : 'C' :   !     : CHAR :
      :                            -------   !          :
      :  M(5)  :  R->(5)   :     : 'H' :   !     : CHAR :
                                   -------   !
                                             !
                                   -------   !
      :  start address     :     :  *>-:--!      :
    R :                            -------         : ROW ARRAY(1:5)CHAR
      :  no. of elements   :     :  5  :
```

ents of M through R; R->(.....) can be used where M(....) is used, e.g.:

```
    24.    R->(1):= 'D';
```

assigns 'D' to M(1). We may, however, put another ROW ARRAY(1:5)CHAR descriptor value into R, so that use of R and M will work differently:

```
25.   R:= <-M(2:4);
```

Here, M(2:4) yields a "ROW ARRAY(1:5)CHAR" value describing the elements 2, 3 and 4 of M. This construct is called a slice. We now have:

```
      Data names                    Data objects      Data modes
      ----------                    ------------      ----------

                                      -------
      :  M(1)  :                      : 'D' :        : CHAR    :
      :                               -------                  :
      :  M(2)  :  R->(1)   :          : 'I' :<--     : CHAR    :
      :                               -------  !               :
    M :  M(3)  :  R->(2)   :          : 'T' :  !     : CHAR    : ARRAY(1:5)CHAR
      :                               -------  !               :
      :  M(4)  :  R->(3)   :          : 'C' :  !     : CHAR    :
      :                               -------  !               :
      :  M(5)  :                      : 'H' :  !     : CHAR    :
                                      -------  !
                                               !
                                      -------  !
      :  start address     :          :  *>-:--!     :
    R :                               -------        : ROW ARRAY(1:5)CHAR
      :  no. of elements   :          :  3  :        :
                                      -------
```

Now, R->(1) is the same element as M(2), R->(2) is M(3), and R->(3) is M(4). R->(4) is illegal, even though M(5) is all right.

3.4.4. Based objects

A based object is an object to which a reference, i.e. the base, is attached. All accessing of the object is done indirectly via the reference.

A based declaration is a declaration having the attribute BASED(<reference location>) following the mode.

Example:

```
1.  NEWMODE CALL_RECORDS = STRUCT( JUNCTION_NO,
2.                                 ANSWER_TIME INT,
3.                                 STANDARD_RATE BOOL);
4.  NEWMODE AREA_SIZE = INT(1:1000);
5.  DCL P REF CALL_RECORDS;
6.      DCL_CALL RECORD_AREA ARRAY(AREA_SIZE) CALL_RECORDS,
7.      CURRENT CALL_RECORDS BASED(P);
```

Line 7 declares CURRENT to be the call record to which P is refering at any instance.

We may for instance initialize CALL_RECORD_AREA by the following statement:

```
 8.  DO FOR I IN AREA_SIZE;
 9.      P := ->CALL_RECORD_AREA(I);
10.      CURRENT := (: 0, 5, TRUE :);
11.  OD;
```

In line 8 I runs through the index range of CALL_RECORD_AREA. In line 9 P is set to refer to each element in turn. In line 10 CURRENT represents the location refered to by P.

3.4.5. Summary

```
<reference mode> ::=
          <bound reference mode>
        | <row mode>
< bound reference mode > ::=
          REF <referenced mode>
<row mode> ::=
          ROW <array mode>
<emptyness literal> ::=
          NULL
<dereferenced bound reference> ::=
          <bound reference expression> ->
<dereferenced row > ::=
          <row expression> ->
<referenced locatiion> ::=
          -> <location>
<based declaration> ::=
          <name list> <mode>
          BASED[(<reference access name>)]
```

3.5. More on composite modes

To realize the full potential of composite modes we must extend the concepts treated earlier in this book.

3.5.1. The do with statement

When making heavy use of structures and field selection it becomes cumbersome to always have to precede the field names by the structure object. This is in particular true when long structure names are used or when the structure is given by an expression. Programs may also become difficult to read.

The do with statement is intended to be used when access is made several times to fields of the same structure. Use of the do with statement may also make the program more efficient.

Example:

Assume the declaration:

```
DCL CALL_RECORD = STRUCT( JUNCTION_NO,
                          ANSWER_TIME INT,
                          STANDARD_RATE BOOL);
```

The do with may be used in manipulating this structure:

```
DO WITH CALL_RECORD;
    JUNCTION_NO := 5000;
    ANSWER_TIME := 2;
OD;
```

This do with statement is equivalent to:

```
CALL_RECORD.JUNCTION_NO := 5000;
CALL_RECORD.ANSWER_TIME := 2;
```

The do with statement effectively opens up the structure, making the field names accessible, without qualification, between the DO - DO brackets.

3.5.2. Variant structures

When defining data structures, there is often the need to describe objects which are similar but not quite identical. The objects usually have some fields in common, while other fields are particular to only some of the objects. Such objects are called variant structures or simply variants. One example of this is a chain of varying length buffers. Each buffer has a pointer to the next one in the chain, while the size of the actual buffer area varies from one buffer to the next.

Consider an example where there is a chain of buffers of two sizes, small and large.

```
 1.  SYN SMALL_SIZE = 4,
 2.      LARGE_SIZE = 12;
 3.  NEWMODE BUFFER_SIZES = SET(SMALL, LARGE);
 4.  NEWMODE BUFFERS = STRUCT(NEXT REF BUFFERS,
 5.                           SIZE BUFFER_SIZES,
 6.                           CASE SIZE OF
 7.                             (SMALL): S_AREA CHAR(SMALL_SIZE),
 8.                             (LARGE): L_AREA CHAR(LARGE_SIZE)
 9.                           ESAC
10.                          );
```

Lines 1 and 2 define the synonyms SMALL_SIZE and LARGE_SIZE, to avoid having the "magic" literals 4 and 12 spread at various places in the program. Line 3 defines a set mode BUFFER_SIZES, the objects of which may assume the values SMALL or LARGE. Lines 4 to 10 defines the mode for the buffers. First each buffer has a reference to the next buffer in the chain. This reference s called NEXT. It is of mode REF BUFFERS, which means that it can refer to either kind of buffer, large or small. Then comes, at line 5, a field called SIZE of mode BUFFER_SIZES. This field is used as a tag in the buffer object to indicate whether the object is a small or a large buffer. The field is called the tag field of the variant.

The CASE - ESAC construct at lines 6 to 9 is called the variant part of the structure. The possible variations are described as alternatives of the case, where the case selector is the name of the tag field used to determine the proper alternative. In this case the buffer area is a string of either 4 or 12 characters. In the former case the field name is called S_AREA, in the latter L_AREA. Field names must be unique also within a variant part. Let us now create a chain of buffer locations.

```
   11.  DCL BUFFER_1 BUFFERS:=(: NULL, SMALL, (SMALL_SIZE)' ' :),
   12.      BUFFER_2 BUFFERS:=(: ->BUFFER_1,LARGE, (LARGE_SIZE)' ' :),
   13.      BUFFER_3 BUFFERS:=(: ->BUFFER_2,SMALL, (SMALL_SIZE)' ' :);
   14.      HEAD REF BUFFERS:= ->BUFFER_3;
```

Lines 11 to 13 declare three buffer object BUFFER_1, BUFFER_2 and BUFFER_3 and chains them together. The tag fields are initialized to SMALL and LARGE and the buffer areas to 4 or 12 spaces respectively. Finally line 14 declares a pointer HEAD, initialized to refer to BUFFER_3.

The created data structure may be pictured as follows:

```
     BUFFER _1     BUFFER _2     BUFFER _3     HEAD

    -----------   -----------   -----------   -----------
    :         :<--:----*    :<--:----*    :<---:----*    :
    :---------:   :---------:   :---------:   -----------
    :  SMALL  :   :  LARGE  :   :  SMALL  :
    :---------:   :---------:   :---------:
    :    :    :   :    :    :   :    :    :
    :---------:   :---------:   :---------:
    :    :    :   :    :    :   :    :    :
    -----------   :---------:   -----------
    :         :   :    :    :   :         :
    :         :   :---------:   :         :
    :         :   :    :    :   :         :
    :         :   :---------:   :         :
    :         :   :    :    :   :         :
    :         :   :---------:   :         :
    :         :   :    :    :   :         :
    -----------   -----------   -----------
```

Now we want to define a procedure which prints the contents of such buffers. One way of doing that is by defining the following procedure PRINT_BUFFER:

```
15.  PRINT_BUFFER:
16.  PROC( BUFFER REF BUFFERS);
17.       CASE BUFFER->.SIZE OF

18.            (SMALL): DO FOR I IN INT(0 : SMALL_SIZE-1);
19.                          OUTCHAR(BUFFER->.S_AREA(I));
20.                     OD;

21.            (LARGE): DO FOR I IN INT(0 : LARGE_SIZE-1);
22.                          OUTCHAR(BUFFER->.L_AREA(I));
23.                     OD;
24.       ESAC;
25.  END PRINT_BUFFER;
```

The procedure takes a reference to either size buffer as a parameter and delivers no result. Lines 17 to 24 is a case statement which determines whether the buffer actually passed as a parameter is small or large. If it is small we print, by means of the procedure OUTCHAR, the 4 characters of its S_AREA in lines 18 to 20. Otherwise it is large and the 12 characters of its L_AREA are printed.

Suppose we want to print the contents of all the buffers in the chain. That can be done by the following statements:

```
26.  DCL CURRENT REF BUFFERS := HEAD;
27.  DO WHILE CURRENT /= NULL;
28.       PRINT_BUFFER(CURRENT);
29.       CURRENT := CURRENT->.NEXT;
30.  OD;
```

Line 26 declares an auxiliary reference called CURRENT, which is to be used as the running buffer reference. It is initialized to refer to the same buffer as HEAD, i.e. BUFFER_3. Lines 27 to 30 are the scanning loop. In line 27 we test whether the end of the buffer chain has been reached. If not the buffer is printed in line 28 and CURRENT is updated to refer to the next buffer in line 29.

3.5.3. Packing

When declaring large amounts of data objects it becomes necessary to utilize the data storage area(s) efficiently. Preferably no more space should be allocated than truly necessary to hold all the values of an object. On many computers, however, such efficiency in storage utilization requires more instructions and more execution time in order to access the data objects. CHILL offers facilities for controlling this trade-off in space/time efficiency by specifying the packing density of data. When nothing is specified, some implementation dependent default packing will be used. Packing is only possible within composite objects.

3.5.3.1. PACK / NOPACK

PACK and NOPACK are components of modes and are used to indicate whether to use a denser or less dense packing than the default. On many computers one of the extremes, e.g. completely unpacked, is default. Thus only the opposite, i.e. PACK, needs to be specified.

Example:

```
    1.  DCL CALL_RECORD STRUCT(JUNCTION_NO INT PACK,
    2.                         ANSWER_TIME INT(0:100) PACK,
    3.                         STANDARD_RATE BOOL PACK);
```

On a 16-bit computer, this object may get the following layout:

```
--------------------------------------
:            JUNCTION _NO            :
:------------------------------------:
:  ANSWER _TIME  :  STANDARD _RATE   :
--------------------------------------
```

If no PACK had be specified the object would probably have required three computer words, instead of now two.

Arrays may also be packed.

Example:

```
    4.  NEWMODE BYTE = INT(0:255);
    5.  DCL X ARRAY(1:100) BYTE PACK;
```

On a 32-bit computer this may result in packing 4 elements to each computer word, requiring a total of 25 words for the entire array X.

3.5.3.2. Layout description

If even more explicit control of data layout is needed, precise layout description may be used. Here the exact positioning of fields and elements within computer words may be specified. This precise layout control may be particulary useful when interfacing to data structures created outside CHILL, e.g. by assembly code modules.

Example:

```
    6.  DCL CALL_RECORD STRUCT(JUNCTION_NO INT POS(0),
    7.                         ANSWER_TIME INT(0:100) POS(1,0:6),
    8.                         STANDARD_RATE BOOL POS(1,7));
```

The position of a field of a structure is specified by the POS attribute. Line 6 says that JUNCTION_NO is to occupy the zeroth word of the structure data area. (Machine words are numbered from zero). The next word is to contain ANSWER_TIME in bits 0 to 6 and STANDARD_RATE in bit 7.

Also the layout of arrays may be specified in detail.

Example:

```
    DCL Y ARRAY(1:100) CHAR
                       STEP(POS(0,1),7);
```

The STEP specification indicates the position of the first element and the no. of bits allotted to each element. Assuming a 36-bit word computer, the example above will pack 5 characters of 7 bits each to the word. The leftmost bit of each word will be unused.

Note that the use of POS and STEP makes the data description highly machine-dependent and very detailed. It should only be used when the power of this specification is really needed. For most applications the default layout or PACK/NOPACK will be sufficient.

3.5.4. Summary

```
<do with statement> ::=
        DO WITH <structure object> {,<structure object>} *;
           <action statement list>
        OD;

<array mode> ::=
        [READ] [ARRAY] (<index set> {,<index set>}*)
        <element mode> {<element layout>}*

<structure mode> ::=
        STRUCT (<fields> {,<fields>}*)

<fields> ::=
        <fixed fields>
      | <alternative fields>

<fixed fields> ::=
        <name list> <mode> [<field layout>]

<alternative fields> ::=
        CASE [<tags>] OF
        <variant alternative> {,<variant alternative>}*
        [ELSE [<variant fields> {, <variant fields>}*]] ESAC
```

```
<variant alternative> ::=
        [<case label specification> {,<case label specification>}*]
        : [<variant fields> {,<variant fields>}*]

<tags> ::=
        <tag field name> {,<tag field name>}*

<variant fields> ::=
        <name list> <mode> [<field layout>]

<element layout> ::=
        PACK | NOPACK | <step>

<field layout> ::=
        PACK | NOPACK | <pos>

<step> ::=
        STEP(<pos> [,<step size> [,<pattern size>]])

<pos> ::=
        POS(<word> [,<start bit> [,<length>]])
      | POS(<word> [,<start bit> : <end bit>])
```

4. Program structure

Program structuring consists of grouping together action and object descriptions which are related. Some reasons why such groupings are desirable might be:

- to modularize a system for easier program development and maintenance, e.g. separate compilation.
- to improve readability and documentation
- to control storage allocation and space/time efficiency
- to isolate program parts from each other, controlling connections between the parts eliminating unwanted side-effects.

The need for proper program structure is most strongly felt in large program systems. Therefore all the facilities of CHILL might not be appreciated on the basis of this introductory manual alone.

The constructs available for program structuring are modules and blocks. Blocks exist in three forms, begin blocks, procedure blocks and process blocks. The following brackets are used for the module and block constructs:

```
- MODULE    END
- BEGIN     END
- PROC      END
- PROCESS   END
```

4.1. Visibility and lifetime

A most important aspect of program structuring is to control the use of names, i.e. where names are declared or defined and where these names may be applied. When a name may be applied at some point in a program, the name is said to be visible at that point.

Example:

```
1.      MODULE
2.          DCL I, J INT:=0;
3.          I:=5;
4.          J:=J+3;
5.      END;
```

The names I and J are declared on line 2 as being objects of mode INTeger and having the initial value zero. In this case both names are visible in lines 3 and 4. Thus it is possible to apply, i.e. use, I and J in the assignment statements of those lines.

An object has a certain lifetime. The lifetime is the time during which the object exists in the program. Some objects may have a permanent lifetime, in which case they exist for the whole duration of the program. The visibility of a name of an object may be smaller than the lifetime of the object itself. This is the same as saying that there may be places in a program where an object may not be used even though it exists.

The lifetime of a location object is important for storage allocation because the lifetime determines when storage must be allocated and when it may be deallocated.

4.2. Blocks

A begin block consists of a list of statements enclosed within brackets BEGIN and END. The block may have a name preceding BEGIN and following END.

Procedures have been presented earlier. Blocks determine both visibility of names and lifetime of objects declared in the blocks.

Blocks may be nested. In this case all names declared in an outer block are automatically visible in all inner blocks, but names declared in inner blocks are never visible in outer blocks.

Names are thus said to be local to the block in which they are declared and global to all inner blocks.

The lifetime of an object is the time during which the block that contains the object declaration is executed.

To illustrate the use of blocks, we will rewrite the list handling example of sec. 3.4.2.

```
1.      CIRCULAR_LIST_2:
2.      MODULE
3.          NEWMODE NODE = STRUCT(PRED, SUC REF NODE, VALUE INT);
4.          DCL POOL ARRAY(1:1000) NODE;
5.          DCL HEAD NODE:=(: NULL, NULL, 0 :);
6.          INSERT:
            PROC(NEW NODE);
7.              DCL TEMPORARY NODE;
8.              TEMPORARY:=NEW;
9.              /* FURTHER ACTIONS TO INSERT NEW ELEMENT
10.                AT START OF THE LIST */
11.         END INSERT;

12.         REMOVE:
            PROC();
13.             /* ACTIONS TO REMOVE ONE ELEMENT FROM
14.                END OF THE LIST */
15.         END REMOVE;

16.         INITIALIZE_LIST:
17.         BEGIN
18.             DCL LAST REF NODE := ->HEAD;
19.             DO FOR NEW IN POOL;
20.                 NEW.PRED := LAST;
21.                 LAST->.SUC := ->NEW;
22.                 LAST:= ->NEW;
23.                 NEW.VALUE:=0;
24.             OD;
25.             HEAD.PRED:=LAST;
26.             LAST->.SUC:= ->HEAD;
27.         END INITIALIZE_LIST;

28.         REMOVE();
29.         REMOVE();
30.         INSERT( (: NULL, NULL, 536 :) );
31.     END CIRCULAR_LIST_2;
```

The outermost level of a program is always a module. Within the module CIRCULAR_LIST_2 the following names are defined:

- a mode NODE
- names of two data objects POOL and HEAD
- two procedures INSERT and REMOVE
- a begin block named INITIALIZE_LIST

There are three blocks nested within CIRCULAR_LIST_2: - INSERT, REMOVE and INITIALIZE_LIST. All the above names defined in CIRCULAR_LIST_2 are visible within these blocks.

In line 6 a formal parameter object NEW of mode NODE is declared, and in line 7 we declare an object called TEMPORARY, of mode NODE. NODE may be used here because it is visible. Both NEW and TEMPORARY are, however, local to INSERT and may never be used outside the procedure. Both their visibility and lifetime are restricted to the procedure INSERT.

The block INITIALIZE_LIST initializes the list structure. For this purpose two auxiliary objects, LAST and NEW are needed. Because they are needed for this purpose only, the initialization has been grouped into a block, to make LAST and NEW unaccessible once this task has been performed.

LAST becomes local to INITIALIZE_LIST. NEW has an even more restricted visibility. It is local to the do loop. Note that the loop counter, named NEW of line 19, is distinctly different from the formal parameter NEW at line 6. They are local to different blocks and are thus names for different objects.

In lines 28 to 30 two elements are removed from the list and one is inserted.

4.3. Modules

Names declared in a module are local to that module. However, global names, i.e. names declared outside the module, are not automatically visible inside the module. Furthermore, local names of a module may be made visible outside the module. Thus a module only controls visibility, not lifetime. The objects declared in a module have a lifetime extending beyond the module, namely that of the first surrounding block.

To make a global name visible within a module, the name must be mentioned in a seize statement. To make a local name visible outside a module, the name must be mentioned in a grant statement.

Example:

```
 1.  M1:
 2.  MODULE
 3.       DCL D INT;

 4.       M2:
 5.       MODULE
 6.            DCL C,E,F INT;
 7.            SEIZE D;
 8.            GRANT C,E;
 9.            /* C,D,E AND F ARE VISIBLE HERE */
10.       END M2;

11.       /* C,D AND E ARE VISIBLE HERE */

12.       M3:
13.       MODULE
14.            DCL G BOOL;
15.            SEIZE E;
16.            /* E AND G ARE VISIBLE HERE */
17.       END M3;

18.       B1:
19.       BEGIN
20.            DCL D BOOL;
21.            /* C AND E ARE ALSO VISIBLE HERE */
22.       END B1;

23.  END M1;
```

In the example above, all objects in modules M2 and M3 have a lifetime extending over the whole program. The visibility of some of the objects are, however, restricted by the modules.

At line 20 another object named D is declared. Thus the D visible in block B1 is distinctly different from the one visible outside, which is the D declared at line 3. We say that the D of line 20 is a redeclaration of the D at line 3.

To give a more practical illustration on the use of modules, let us rewrite the list handling program once more.

The problem with CIRCULAR_LIST_2 is that the data structure of the list is visible everywhere in the module. Thus it is possible to manipulate the list directly, without using the procedures INSERT and REMOVE. This opens the possibility for introducing dangerous shortcuts in the program, destroying its modularity.

This possibility is eliminated if the module is rewritten in the following way:

```
CIRCULAR_LIST_3:
MODULE

     HANDLE_LIST:
     MODULE
          GRANT INSERT, REMOVE, NODE;
          NEWMODE NODE=STRUCT(PRED, SUC REF NODE, VALUE INT);
          DCL POOL ARRAY(1:1000)NODE;
          DCL HEAD NODE:=(: NULL,NULL,0 :);

          INSERT:
          PROC(NEW NODE);
               /* INSERT ACTIONS */
          END INSERT;

          REMOVE:
          PROC();
               /* REMOVE ACTIONS */
          END REMOVE;

          INITIALIZE_LIST:
          BEGIN
               DCL LAST REF NODE:= ->HEAD;
               DO FOR NEW IN POOL;
                    NEW.PRED := LAST;
                    LAST->.SUC:= ->NEW;
                    LAST:= ->NEW;
                    NEW.VALUE:=0;
               OD;
               HEAD.PRED:=LAST;
               LAST->.SUC:= ->HEAD;
          END INITIALIZE_LIST;

     END HANDLE_LIST;

     DCL NODE_A NODE:=(: NULL,NULL,536 :);
     REMOVE();
     REMOVE();
     INSERT(NODE_A);
END CIRCULAR_LIST_3;
```

The only names, concerning the list handling visible in the outer module are now INSERT, REMOVE and NODE. Thus it is possible to declare new NODE objects, like NODE_A. Handling of the list, however, can only be done by INSERT and REMOVE because the names HEAD and POOL are not visible outside the module HANDLE_LIST.

4.4. Summary

```
<begin-end block> ::=
        BEGIN <begin-end body> END;
<module> ::=
        MODULE <module body> END;
<begin-end body> ::=
        <data statement list> <action statement list>
<module body> ::=
        {<data statement> | <visibility statement>}*
        <action statement list>
<visibility statement> ::=
        <grant statement>
      | <seize statement>
<grant statement> ::=
        GRANT <name list> ;
<seize statement> ::=
        SEIZE <name list> ;
```

4.5. Concurrent execution

Concurrencymic program property, i.e. something occurring during the execution of a program. Concurrent execution occurs when various parts of a program are executed at the same time, i.e. in parallel. "At the same time" is seen from the logical point of view of the program. It does not necessarily mean that it physically occurs at the same time.

A telephone exchange is a very typical concurrent execution system. For instance telephony devices are working in parallel with the central processing unit and many calls are in various stages of progress at the same time.

4.5.1. The process concept

The unit of concurrent execution is the process. A process is a dynamic program part which may be executed concurrently with other processes of the program. Thus an executing CHILL program consists of one or more processes. Within a process, execution is always serial.

A process is, at the logical program level, always in one of three states: nonexistent, active or delayed. Transitions between these states are caused by various actions of the program. Thus the lifecycle of a process can be pictured as shown below.

```
               ---------------
               :             :
               : nonexistent :
               :             :
               ---------------
                      :
                      :   starting
                      :
                      V
               ---------------        delaying        ------------
               :             :------------------->:           :
               :   active    :                    : delayed   :
               :             :<-------------------:           :
               ---------------       continuing       ------------
                      :
                      :   stopping
                      :
                      V
               ---------------
               :             :
               : nonexistent :
               :             :
               ---------------
```

4.5.1.1. The process definition statement

A process definition is a template from which processes are created. The process definition describes the actions of a process and its local objects. The process definition is manifest in a CHILL source program as a process definition statement.

This statement is syntactically similar to the definition statement. The word PROCESS is used instead (

Example:

```
    LOCAL_CALL:
    PROCESS(A_PART SUBSCRIBERS);
         ------
         ------
    END LOCAL_CALL;
```

A process definition statement is a block.

Process definitions are not allowed inside blocks, e g. begin-end blocks or procedure blocks.

A CHILL program is considered to consist of one language defined outer process and any number of user defined ones. The definition of the outer process is considered to consist of all statements not enclosed in explicit process definition statements.

4.5.1.2. The start statement

From a process definition we may create new processes. Many processes may be created from the same process definition. This creation is performed by the start statement. This statement consists of the word START followed by the process definition name and possibly an actual parameter list.

Example:

```
    START LOCAL_CALL (ACTIVE_LINE1, 200, 5);
    START LOCAL_CALL (ACTIVE_LINE2, 50, 2);
```

Each line above creates a new process. After the second line these two processes are executed concurrently. The actual parameter list of the start statement may contain more parameters than the formal parameter list of the process definition. The meaning of such extra parameters is implementation dependent. One use of such parameters is to indicate the amount of recources given to the process. In the example above these resources are assumed to be the size of the data area of the process and its priority.

The outer process is started by an implicit start statement which thus initiates execution of the CHILL program.

4.5.1.3. The stop statement

In order for a process to stop its execution it must execute a stop statement. This statement always occurs implicitly at the end of a process definition. If stopping is required at any other point, an explicit stop statement must be written. It consist simply of the word STOP.

Example:

```
    CALL_HANDLER:
    PROCESS(LINE_NO INT);
        DO FOR_EVER;
           ------
           ------
           IF LINE_NO > 10000 THEN STOP; FI;
        OD;
    END CALL_HANDLER;
```

Note that a STOP statement may be placed within a procedure. In this case, the effect is to stop the process calling the procedure.

4.5.1.4. Instance modes

So far we have had no means of identifying or getting hold of processes. Once they were started they would run their own course. Process instance values are used to identify processes. Process instance locations are declared like any other location object using the special mode INSTANCE.

Process instance values may be produced in several ways. First of all there is the start expression which delivers a value identifying the process that has just been started.

Example:

```
    DCL CALL1 INSTANCE;
    CALL1 := START LOCAL_CALL(ACTIVE_LINE, 200, 5);
```

It is now possible to reach the process via the process instance value in CALL1.

To get to know the identity ot the current process, we use the operator THIS. THIS yields the instance value of the process executing the THIS operator. A process instance value which should not identity any process is said to be the value NULL.

Example:

```
1.      DCL CALL1,CALL2 INSTANCE := NULL;

2.      CALL_HANDLER:
3.      PROCESS(LINE_NO INT);
            - - - - -
            - - - - -
4.          CALL2 := THIS;
            - - - - -
5.      END;

6.      CALL1 := START CALL_HANDLER(2, 200, 5);
7.      START CALL_HANDLER(10, 200, 5);
```

After executing line 4 caused by the starting at line 6, CALL1 and CALL2 will identify the same process. When line 4 is executed, caused by line 7, however, CALL2 will identify another process different from the one identified by CALL1. It is here assumed that the first process has executed beyond line 4 before the second process starts.

4.5.1.5. Summary

```
<process definition statement> ::=
        <name> : <process definition>
        [<handler>] [<process name>];

<process definition> ::=
        PROCESS ([<formal parameter list>]);
        <process body> END

<start statement> ::=
        <start expression> ;

<start expression> ::=
        START <process name> ([<actual parameter list>])

<actual parameter list> ::=
        <actual parameter> {,<actual parameter>}*

<stop statement> ::=
        STOP;

<instance mode> ::=
        INSTANCE
```

4.5.2. Coordination between processes

If several processes are required to accomplish a common task, there is a need for them to cooperate. This means that the processes must be coordinated to exchange information at certain points.

Another need for coordination arises when several processes are competing for the use of a common resource. Then the processes must be scheduled in such a way that only one process gets to use the resource at any one time.

CHILL provides several methods for realizing process coordination. The methodes differ mainly in the degree of explicit control exerted by the programmer. In general, care should be taken not to mix the various methods within one subsystem.

CHILL distinguishes between three aspects of process coordination. Synchronization means to coordinate processes such that their execution is performed in a lockstep fashion, i.e. one process may have to be delayed for another process to reach a certain point before the former can continue.

Communication is the exchange of messages, i.e. data objects, between processes. Communication implies synchronization.

Mutual exclusion is used to ensure safe access to common resources. Mutual exclusion also implies synchronization.

4.5.2.1. Event modes

Event mode objects and their related operations are used to provide synchronization between processes. Event objects are declared by using the mode EVENT.

It is possible to DELAY a process to make it wait for an event to occur and a process may cause an event such that DELAYed processes are able to CONTINUE. This is done by the DELAY and CONTINUE statements respectively. They consist of these words followed by an event location denotation as operand. The example below will illustrate the use of events.

```
1.      SWITCH_BOARD:
2.      MODULE
3.          DCL OPERATOR_IS_READY EVENT;

4.          CALL_DISTRIBUTOR:
5.          PROCESS();
6.              DO FOR EVER;
7.                  WAIT(10 /*seconds*/);
8.                  CONTINUE OPERATOR_IS_READY;
9.              OD;
10.         END CALL_DISTRIBUTOR;

11.         CALL:
12.         PROCESS();
13.             DELAY OPERATOR_IS_READY;
14.         END CAR;

15.         START CALL_DISTRIBUTOR();
16.         DO FOR I IN INT(1:100);
17.             START CALL();
18.         OD;
21.     END SWITCH_BOARD;
```

This example illustrates a switch board which queues incoming calls and feeds them to the operator at an even rate. Every time the operator is ready one and only one call is let through. This is handled by a call distributor which lets calls through at fixed intervals. If the operator is not ready or there are other calls waiting, a new call must queue up to wait for its turn.

Line 3 declares OPERATOR_IS_READY to be the event controlling the distribution. The CALL_DISTRIBUTOR at lines 4 to 12 defines processes that cause this event every 10 seconds. WAIT at line 7 is supposed to be a procedure programmed elsewhere. Line 8 should be read: "Allow one of the processes, DELAYed until the ..ERATOR_IS_READY, to CONTINUE if there is one present".

CALLs are defined at lines 11 to 14 to be processes which are DELAYed until the OPERATOR_IS_READY and thereafter just stopped.

When executing this program, the following happens. The whole module SWITCH_BOARD except the lines 4 to 14 defines the outer process. It is started and executes its first action statement at line 15. Here one CALL_DISTRIBUTOR process is started.

Lines 16 to 18 start 100 CALL processes which queue up and are let through to the operator one by one. Once these processes are started, the outer process is suspended. The CALLs stop one by one as they are let through the gate, while the CALL_DISTRIBUTOR keeps going forever.

Notice that if the OPERATOR_IS_READY when no CALLs are waiting, the event causes no action. We say that events are non-persistent.

4.5.2.2. The delay case statement

The delay statement of the previous section, allows a process to be delayed waiting for only one event. The delay case statement allows a process to wait for one of a number of events. An example will illustrate the use of the delay case statement.

Example:

In continuing our switch board example above, suppose the operator closes at 5 p.m. To handle this we declare another event

```
    DCL SWITCH_IS_CLOSED EVENT;
```

The definition of the operator could be:

```
OPERATOR:
PROCESS();
    DO FOR_EVER;
        IF TIME = 1700
            THEN
                CONTINUE SWITCH_IS_CLOSED;
        FI;
    OD;
END OPERATOR;
```

The CALL definition can now be changed to:

```
CALL:
PROCESS();
        DELAY CASE
        (OPERATOR_IS_READY): /* some action */ ;
        (SWITCH_IS_CLOSED): DO FOR I IN INT(1:100);
                            CONTINUE OPERATOR_IS_READY;
                            /*empty the queue*/
                            OD;
        ESAC;
END CALL;
```

CALLs now are delayed until either of the events occur, and action is taken for whichever event occurs first. The first call process to detect that the SWITCH_IS_CLOSED will empty the queue by letting all the possibly waiting calls through to the operator as fast as possible. For the sake of simplicity we assume that no call will arrive after closing time.

When one of the possible events have occured, the process is no longer delayed for any of the other events. If both events should happen at the same time, one of them is selected arbitrarily.

4.5.2.3. Buffer modes

Buffer mode objects and their operations are used to provide communication between processes. Buffer location objects are declared with the word BUFFER followed by a mode for the message.

Messages can be sent to and received from buffers by processes. This is done by the SEND and RECEIVE statements respectively.

Example:

```
1.  MODULE
2.          NEWMODE INPUT = SET(ANSWER, CLEAR_BACK, RELEASE);
3.          DCL MAIL_BOX BUFFER(10) INPUT;

4.          RECEIVER:
5.          PROCESS();
6.              DCL MESSAGE INPUT;

7.              MESSAGE := RECEIVE MAIL_BOX;
8.              SEND MAIL_BOX(RELEASE);
9.          END RECEIVER;

10.         START RECEIVER();
11.  END;
```

In line 2 we define a mode for the messages which are to be sent. In line 3 MAIL_BOX is declared to be a BUFFER which can store up to 10 messages of mode INPUT.

RECEIVER is a process definition at lines 4 to 9. First a local object MESSAGE is declared. Then at line 7 RECEIVE will delay the process until a MESSAGE is present in the MAIL_BOX. When this is the case, the message in the buffer is assigned to MESSAGE and the process continues.

At line 8 the process sends the message RELEASE to the buffer MAIL_BOX. Presumeably this message will be received from the buffer by another process.

A buffer is <u>persistent</u> in contrast to events, i.e. a message sent for which there are no receivers at the moment, will be stored in the buffer until it is received by someone. The sending process is, however, not delayed awaiting reception.

4.5.2.4. Signals

Signals are used to provide both synchronization and communication between processes. Signals are defined in a special signal definition statement.

Example:

```
    SIGNAL T;
    SIGNAL S = (INT, BOOL);
```

The first line defines a signal T without a message part, i.e. the signal can be used for synchronization only. The second line defines a signal S with a message part consisting of an INT and a BOOL value. Thus the message part of a signal definiton can consist of a list of mode denotations enclosed in parentheses.

When used with signals, the send statement may have a trailing to-clause giving the process instance of the process to which the signal is sent.

Example:

```
    SEND S(5, TRUE) TO PI;
    SEND T;
```

In the first line, the signal S is sent to the process identified by the process instance PI. In the second line, no specific receiver is given. Then the signal may be received by any process.

A signal persistent.

4.5.2.5. The receive case statement

The receive case statement allows for receiving any one of a set of buffers or signals. It is similar to the delay case statement but with facilities added to handle the message part of buffers or signals.

Example:

To illustrate the use of signals and the receive case, (buffers might have been used instead) we will look at an example where an ALLOCATOR manages a set of resources, in this case a set of COUNTERs. The module is part of a larger system where there are USERs, that can request the services of the COUNTER_MANAGER. The module is made to consist of two process definitions, one for the ALLOCATOR and one for the COUNTERS. This can be pictured as shown below.

```
                  ---------------
                  :             :
 ACQUIRE          :             :
   ------->:      :             :
                  :  ALLOCATOR  : CONGESTED
                  :             :------->
                  :             :
 RELEASE          :             :
   ------->:      :             :
                  :             :
                  ---------------
                       :   :
             INITIATE  :   :  TERMINATE
                       :   :
                       :   :
                       V   V
                  ---------------
                  :             :
                  :             :    READY
 STEP             :             :------>
   ------->:      :   COUNTER   :
                  :             :
                  :             :    READOUT
                  :             :------>
                  :             :
                  ---------------
```

INITIATE and TERMINATE are internal signals sent from the ALLOCATOR to the COUNTERs. All the other signals are external, being sent from or to the USERs.

```
<> FREE STEP;
COUNTER_MANAGER:
MODULE
     SEIZE /* external signals */
          ACQUIRE, RELEASE, CONGESTED,STEP,READOUT;
     SIGNAL INITIATE = (INSTANCE),
          TERMINATE;

     ALLOCATOR:
     PROCESS
          NEWMODE NO_OF_COUNTERS = INT(1:10);
     DCL COUNTERS ARRAY (NO_OF_COUNTERS)
                    STRUCT (COUNTER INSTANCE,
                              STATUS SET (BUSY,IDLE));
     DO FOR EACH IN COUNTERS;
     EACH:= (: START COUNTER(), IDLE :);
     OD;

     DO FOR EVER;
     BEGIN
          DCL USER INSTANCE;
          AWAIT_SIGNALS:
          RECEIVE CASE SET USER;
```

```
22.     (ACQUIRE):
23.         DO FOR EACH IN COUNTERS;
24.         DO WITH EACH;
25.           IF STATUS = IDLE
26.           THEN
27.             STATUS:=BUSY;
28.             SEND INITIATE (USER) TO COUNTER;
29.             EXIT AWAIT_SIGNALS;
30.           FI;
31.         OD;
32.         OD;
33.         SEND CONGESTED TO USER;

34.         (RELEASE IN THIS_COUNTER);
35.         SEND TERMINATE TO THIS_COUNTER;
36.         FIND_COUNTER:
37.         DO FOR EACH IN COUNTERS;
38.         DO WITH EACH;
39.           IF THIS_COUNTER = COUNTER
40.           THEN
41.             STATUS:= IDLE;
42.             EXIT FIND_COUNTER;
43.           FI;
44.         OD;
45.         OD FIND_COUNTER;
46.     ESAC AWAIT_SIGNALS;
47.     END;
48.     OD;
49.     END ALLOCATOR;

50.     COUNTER:
51.     PROCESS();
52.         DO FOR EVER;
53.         BEGIN
54.             DCL USER INSTANCE;
55.                 COUNT:= 0;
56.             RECEIVE CASE
57.                 (INITIATE IN RECEIVED_USER):
58.                     SEND READY TO RECEIVED_USER;
59.                     USER:= RECEIVED_USER;
60.             ESAC;
61.             WORK_LOOP:
62.             DO FOR EVER;
63.                 RECEIVE CASE
64.                     (STEP): COUNT +:=1;
65.                     (TERMINATE):
66.                         SEND READOUT(COUNT) TO USER;
67.                         EXIT WORK_LOOP;
68.                 ESAC;
69.             OD WORK_LOOP;
70.     END;
71.     OD;
72.     END COUNTER;

73.     START ALLOCATOR();
74.  END COUNTER_MANAGER;
```

In line 1 there is a compiler directive to free the reserved word

STEP to allow it to be used as a user defined name. COUNTER_MANAGER is the module containing all the facilities for handling the counters.

In lines 4 and 5 we seize the external signals, and in lines 6 and 7 we define the internal ones. INITIATE has a message part consisting of an INSTANCE value.

Lines 8 to 49 is the definition of ALLOCATOR processes. Line 10 defines the number of counters available. Lines 11 and 12 declares a resource list which is an array used to keep track of the state of the COUNTERS. Each element of the array is a structure countaining an instance identifying the COUNTER process and a status showing whether the counter is busy or idle.

Lines 14 to 16 comprise a loop for starting all COUNTER processes.

Lines 17 to 48 form an eternal loop enveloping the receive case statement labelled AWAIT_SIGNALS at lines 20 to 46. The receive case has two alternatives. It can receive the signal ACQUIRE at line 22 and RELEASE at line 34. When a signal is received, the identity of the sender is stored in the object USER at line 21. In case ACQUIRE is received, we go through the COUNTERS array to see if there is an idle counter. If there is, it is marked BUSY and an INITIATE signal is sent to the counter at line 28. Then we exit the receive case, whereby we loop around and wait at line 21 for the next signal to arrive. If no counter is wailable, the signal CONGESTED is returned to the user at line 33.

If RELEASE is received at line 34, we get the identity of the counter to be released in the object THIS_COUNTER.

Line 35 asks the counter to terminate itself. Then we must locate this counter in the resource list and change its status to IDLE. This is done in lines 36 to 45.

Lines 50 to 72 comprise the definition of COUNTER processes. Again there is an eternal loop, lines 52 to 71.
The count to be stepped is declared at line 55 and initialized to zero.

Lines 56 to 60 is a receive case which can receive only one signal, INITIATE, with a message identifying the user wanting to acquire the counter. The READY signal is sent to this user at line 58, and then the identity of the user is saved for later use.

The lines 61 to 69 comprise the main working loop. The receive case at lines 63 to 68, can receive either a STEP signal or TERMINATE.

In the first case the count is stepped at line 64. In the second case we send the READOUT signal with the COUNT message to the USER at line 66 and exit the work loop. Thereby the outer loop will take control back to line 56 where the process waits for another initiatiation.

At line 73 we have the statement starting the ALLOCATOR. This is the first statement of the COUNTER_MANAGER to be executed. Once the ALLOCATOR is running, it in turn starts the COUNTERs and the COUNTER_MANAGER is ready to receive signals from the outside.

4.5.2.6. Mutual exclusion

Mutual exclusion of processes competing for the same objects may be programmed with the tools shown in previous chapters. However, CHILL also provides a special construct, the region, which is safer and more convenient to use.

A region looks like a module. It starts with the opening bracket REGION. Inside, however, it may contain only definition and declaration statements.

The common objects for wich the processes are competing, are declared local to the region and may not be granted outside. The common objects may only be accessed by region procedures. These procedures are also defined local to the region but are granted outside.

When a process calls a region procedure, another process, calling the same or another of the procedures of the same region, is delayed until the first process has completed its procedure. Thus only one process at a time may access the common objects declared inside the region.

Regions must not be nested inside other regions or inside blocks. Region procedures may not call other region procedures neither directly nor indirectly via other procedures.

Example:

We describe a region controlling the allocation of ten resources. The units can be allocated and deallocated one at a time. If no resources are available, the requesting process is delayed until some other process frees a resource.

```
1.  ALLOCATE_RESOURCES:
2.  REGION
3.      GRANT ALLOCATE, DEALLOCATE;
4.      NEWMODE RESOURCE_SET = INT(0:9);
5.      DCL ALLOCATED ARRAY(RESOURCE_SET) BOOL :=
                      (: (RESOURCE_SET): FALSE :);
6.      DCL RESOURCE_FREED EVENT;

7.      ALLOCATE:
8.      PROC()(INT);
9.          DO FOR EVER;
10.             DO FOR I IN RESOURCE_SET;
11.                 IF NOT ALLOCATED(I)
12.                 THEN
13.                     ALLOCATED(I) := TRUE;
14.                     RETURN I;
15.                 FI;
16.             OD;
17.             DELAY RESOURCE_FREED;
18.         OD;
19.     END ALLOCATE;

20.     DEALLOCATE:
21.     PROC(I INT);
22.         ALLOCATED(I) := FALSE;
23.         CONTINUE RESOURCE_FREED;
24.     END DEALLOCATE;

25.  END ALLOCATE_RESOURCES;
```

Line 3 grants the procedures ALLOCATE and DEALLOCATE to the outside. In line 4 the set of resources RESOURCE_SET is defined with the resources numbered from 0 to 9. Then line 5 declares a boolean array ALLOCATED containing one flag for each resource in the set. The array is initialized to all FALSE, i.e. no resourses are allocated initially.

Line 6 declares the event RESOURCE_FREED for which a process must wait in case no resources are available. Line 7 to 19 is the procedure ALLOCATE. The loop at lines 10 to 16 scans the ALLOCATED array to find a free resource. In that case the resource is taken at line 13 and the procedure returns the index of the allocated resource. If no resource is available, line 17 is executed delaying the process until a resource is freed. When that happens we loop back to line 9 for another scan off ALLOCATED.

DEALLOCATE at lines 20 to 24 sets the proper element of ALLOCATED to FALSE and indicates that fact at line 23 in case other processes have been delayed waiting for the resource to become available.

4.5.2.7. Summary

```
<synchronization mode> ::=
        <event mode>
      | <buffer mode>

<event mode> ::=
        EVENT

<buffer mode> ::=
        BUFFER [(<buffer length>)] <buffer element mode>

<signal definition statement> ::=
        SIGNAL <name> [= (<mode> {,<mode>}*)];

<continue statement ::=
        CONTINUE <event location>;

<delay statement ::=
        DELAY <event location> [,<priority>];

<delay-case statement ::=
        DELAY CASE [SET <instance location>;]
        {<delay alternative>}*
        ESAC;

<delay alternative> ::=
        (<event list>):<action statement list>

<event list> ::=
        <event location> {,<event location>}*

<send statement> ::=
        <send signal statement>
      | <send buffer statement>

<send buffer statement ::=
        SEND <buffer location>(value>) [,<priority>];

<send signal statement> ::=
        SEND <signal name> [(value> {,<value>}*)]
        [,<priority>] [TO <instance expression>];

<receive expression> ::=
        RECEIVE <buffer location>

<receive-case statement> ::=
        <receive-signal-case statement>
      | <receive-buffer-case statement>

<receive-buffer-case statement> ::=
        RECEIVE CASE [SET <instance location>;]
        {<buffer receive alternative>}*
        [ELSE <action statement list>]
        ESAC;

<buffer receive alternative> ::=
        (<buffer location> IN <name>) : <action statement list>
```

```
<receive-signal-case statement> ::=
        RECEIVE CASE [SET <instance location>]
        {<signal receive alternative>}*
        [ELSE <action statement list>]
        ESAC;

<signal receive alternative> ::=
        (<signal name> [IN <name list>]):<action statement list>

<region> ::=
    [<name>:] REGION <region body>
            END [<handler>] [<region name>];

<region body> ::=
        {<declaration statement> | <definition statement> |
        <visibility statement>}*
```

4.6. Exception handling

An exception is a condition which should not occur during normal execution of a program. Which conditions constitute "normal" respectively "exceptional" execution depends much on the viewpoint of the programmer when a program is constructed. One use of exceptions is to handle conditions which often are termed "errors". Typical examples are arithmetic OVERFLOW, EMPTY pointer access etc. Another use of exceptions is to handle infrequently occuring conditions. The exception handling constructs can, however, also be used to an advantage to handle other cases of complex flow of control.

Exceptions are CAUSEd to occur. When they do, control is transfered to appropriate handlers.

Example:

```
1.  DCL A INT(0:256), B,C INT:=0;
2.  A:=B+C ON
3.          (OVERFLOW):  OUTTEXT('OVERFLOW');
4.          (RANGEFAIL): OUTTEXT('RANGEFAIL');
5.          ELSE         OUTTEXT('OTHER ERROR');
6.         END;
```

The addition on line 2 may cause an arithmetic overflow. In that case control goes to line 3 where an appropriate message is printed. The execution continues after line 6. The assignment at line 2 may detect that the result of B+C is outside the allowed range for A. If that occurs, control goes to line 4 and thereafter following line 6. If any other exception should occur, control goes the else alternative at line 5.

4.6.1. Handlers

A handler for an exception is a piece of the program to which control is transfered when the exception is caused. A set of handlers may be appended to any action statement including the initialization part of declaration statements, and to procedure and process definition statements.

A set of handlers is bracketed by the words ON and END. Each handler consists of the parenthesized exception name followed by a colon, followed by an action statement list. See lines 2 to 6 in the preceeding example. The handler set may also contain an else-alternative which is the handler of any non-mentioned exceptions.

4.6.2. Causing exceptions

Exceptions may be caused explicitly or implicitly. Explicite causing is effected by executing a cause action. It consists of the word CAUSE followed by the exception name. Implicite causing is effected by various language constructs the execution of which may give rise to exceptional conditions, e.g. OVERFLOW.

The set of exceptions which can be caused implicitly is predefined in CHILL. They are listed below. Predefined exceptions may also be cause explicitly.

Exception name	Caused by
EMPTY	Dereferencing an empty reference, calling an empty procedure location or sending to an empty process instance location. Taking MAX or MIN of an empty powerset.
MODEFAIL	Mode errors which cannot be detected at compile time. This includes improper lengths of strings and slices, including checking the length of the significant part of a varying string. Dereferencing a mode-free reference with improper mode specified. Using improper version of parameterized structuremodes. Calling a procedure recursively which has been specified as non recursive.
TAGFAIL	Applying fields of variant structures with improper tag fields.
SPACEFAIL	Call to GETSTACK or entering a block when storage requirements cannot be satisfied.
RANGEFAIL	PRED or SUCC on references to array elements when the reference goes out of the bounds of the array. Indexing, substringing or slicing outside the

	array or string boundaries. Value out of range. Illegal priority values. Negative step values.
DELAYFAIL	The event queue is exceeded.
ASSERTFAIL	The assert condition delivers FALSE.
OVERFLOW	Overflow in arithmetic operations and in SUCC or PRED operations.

4.6.3. Formal exceptions

When handling exceptions occurring inside a procedure, it is sometimes desireable to handle them at the calling point rather than at the end of the procedure definition.

For this purpose formal exceptions are provided. The list of formal exceptions must be stated in the procedure definition and becomes a part of the procedure mode.

Example:

```
1.  MODULE
2.       P: PROC()() EXEPTION (CONGESTION, TIMEOUT);
3.       IF /* some condition */ THEN CAUSE ABORT; FI;
4.       IF /* some other condition */
5.           THEN CAUSE CONGESTION;
6.           ELSE CAUSE TIMEOUT;
7.       FI;
8.       END ON
9.       (ABORT): /* some action */;
10.      END P;

11.      P() ON
12.      (CONGESTION): /* actions */;
13.      (TIMEOUT): /* actions */;
14.      END;
15. END;
```

At lines 2 to 8 there is a procedure definition the execution of which may cause three exceptions: CONGESTION, TIMEOUT and ABORT. The choice has been made to handle ABORT locally by the handler at lines 8 to 10. The two others are to be handled at the point of call at lines 11 to 14. When ABORT is caused, control goes to line 9. Thereafter the procedure returns normally. When CONGESTION or TIMEOUT are caused, the procedure is terminated and control is returned to the handlers at lines 12 or 13 respectively.

4.6.4. Searching for handlers

The appropriate handler for an exception is found in the following way. If a handler for the exception is appended to the statement which caused the exception, that handler is excecuted. Otherwise there is a search outwards through the nesting levels to see if a handler is appended to some enclosing bracketed statement, or if there is a formal exception by that name. In the latter case control is transferred to the calling point at which place the search continues outwards again.

If an exception, for which there is no handler, occurs in a procedure or process body, that procedure or process is terminated abnormally.

4.6.5. Summary

```
<handler> ::=
      ON {<on-alternative>}+
      [ELSE <action statement list>] END

<on-alternative ::=
      (<exception list>) : <action statement list>

<cause statement> ::=
      CAUSE <exception name>;
```

5. Optional syntaxes

In some cases CHILL allows for more than one way of writing constructs with the same meaning. For pedagogical reasons only one syntax has been selected for the main part of this manual.

In this chapter the other syntaxes are shown and examples provided. The alternate syntaxes may be substituted for the ones used elsewhere in this manual. The examples used in this section are transcribed from the sections where the features are introduced, and may thus easily be substituted for the orginal ones. The reader is referred to those sections for more explanation of the examples.

The syntaxes are described by means of syntax rules. The syntax mainly used in this manual is given as first alternative, while the equivalent syntaxes are shown as alternatives.

5.1. Assignment symbol

```
<assignment symbol> ::=
        := | =
```

Example:

```
        I = 1;
        I = -2;
        I = (I+2)*J;
        I,J,K = 0;
```

5.2. Integer modes

```
<integer mode> ::=
        INT | BIN
<range mode> ::=
        RANGE (<literal range>)
      | BIN (<integer literal expression>)
```

The mode denotation BIN(n) is equivalent to INT(0:2**n-1). "**" is used to mean "to the power of". This means that an object of mode BIN(n) will require at least n bits to be represented on a binary computer.

Example:

```
        NEWMODE BYTE = BIN(8); /* SAME AS INT(0:255) */
        DCL CODE BIN(3);       /* SAME AS INT(0:7)   */
```

5.3. Array mode denotation

```
<array mode> ::=
       [ARRAY] (<index set>) <element mode>
     | [ARRAY] (<index set> {,<index set>}*) <element mode>
```

The word ARRAY is optional in the array mode denotation. For arrays of arrays, i.e. multidimensional arrays, the mode denotation may be written with a comma between the index sets of each dimension. Thus the following examples are equivalent mode denotation for arrays of 3 by 5 integers:

```
    SYNMODE MATRICES1 = ARRAY(1:3) ARRAY(1:5) INT,
            MATRICES2 = (1:3)(1:5) INT,
            MATRICES3 = ARRAY(1:3, 1:5) INT,
            MATRICES4 = (1:3, 1:5) INT;
```

5.4. Indexing multidimensional arrays

```
<array element> ::=
       <array location> ( <expression>)
     | <array location> (<expression> {, <expression> }*)
```

In the first alternative the array location includes subarrays, while in the latter only entire arrays are allowed.
Assuming the declaration:

```
     DCL MATRIX (1:3, 1:5) INT;
```

The following are equivalent ways of indexing MATRIX:

```
       MATRIX(2)(4)

       MATRIX(2, 4)
```

5.5. Level-numbered structure mode denotation

```
<structure mode> ::=
       <nested structure mode>
     | <level structure mode>

<nested structure mode> ::=
       STRUCT (<fields> {, <fields>}*)

<level structure mode> ::=
       1 [<array specification]
         <(2) <level fields> {, <(2) level fields>}*

<(n) level fields> ::=
       n <name list> <mode>
     | n <name list> [<array specification>]
       <(n+1) level fields> {, <(n+1 level fields>}*
```

The level-numbered notation provides for indicating nesting levels within a structure by means of level numbers, instead of parentheses.

Examples:

```
    Level-numbered notation          Nested notation

    DCL 1 CALL_RECORD,               DCL CALL_RECORD STRUCT
          2 JUNCTION_NO,                 (JUNCTION_NO,
            ANSWER_TIME BIN,               ANSWER_TIME BIN,
          2 STANDARD_RATE BIT(1);          STANDARD_RATE BIT(1));

    DCL 1 X,                         DCL X STRUCT
          2 A(1:3) BIN,                  ( A(1:3) BIN,
          2 B,                             B STRUCT
            3 C BOOL,                        ( C BOOL,
            3 D BOOL,                          D BOOL)),

        1 Y(1:5),                        Y(1:5) STRUCT
          2 E BIN,                               ( E BIN,
          2 F BOOL,                                F BOOL,
          2 G CHAR;                                G CHAR);
```

5.6. Procedure call statement

```
<procedure call statement> ::=
        <procedure call>;
      | CALL <procedure call>;

<procedure call > ::=
        <procedure name> (<actual parameter>{, <actual parameter>}*)
```

In procedure call statements the word CALL may precede the procedure call. Note that CALL may not be used when a procedure delivering a value is called within an expression.

Example:

```
        CALL ANALYZE_DIGIT(CURRENT_DIGIT);
        CALL CHECK();
```

but

```
        R:=5*MAX(K+L,M)+3;
```

5.7. Result specification in procedure definition

```
<result specification> ::=
        (<mode>)
      | RETURNs (<mode>)
```

Example:

```
        MAX: PROC(I, J INT) RETURNS(INT);
                  IF I>J
                  THEN
                       RESULT I;
                  ELSE
                       RESULT J;
                  FI;
        END MAX;
```

5.8. Referencing

```
<referenced location> ::=
        -> <location>
      | ADDR (<location>)
```

Examples:

```
        DCL I INT:=2,
            J INT:=3,
            RI REF INT := ADDR(I);
            J:=RI-> +4;
            RI:=ADDR(J);
            I:=I*RI->;
```

6. Appendix A. Comment and evaluation sheet

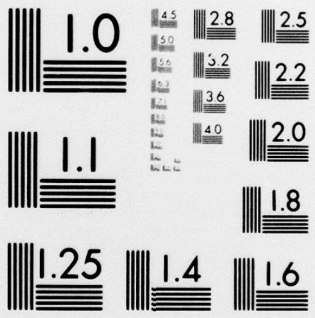
In order to develop this manual to best suit your needs, we kindly ask you to supply your comments, corrections, suggestions for additions etc. Please write them down on this form and mail it to

RUNIT
attn.: Kristen Rekdal
N-7034 Trondheim-NTH
Norway

Use additional sheets, if necessary. Please be specific whenever possible.

Submitted by: ------------------------------

INDEX LIST



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A